目錄

Node.js API 中文文档

译者

DavidCai1993

贡献

项目 github repo

欢迎纠错，欢迎发Pull request，同时也欢迎star，欢迎follow。

贡献者

> 排名不分先后

- jnduan
- caizixian
- Richardwei

License

The MIT License (MIT)

Assert

稳定度: 2 - 稳定

本模块被用来为你的应用编写单元测试，你可以通过 `require('assert')` 来使用它。

assert.fail(actual, expected, message, operator)

抛出一个打印实际值 `actual` 和期望值 `expected` 的异常，使用分隔符 `operator` 隔开。

assert(value[, message]), assert.ok(value[, message])

测试 `value` 是否为真，它等同于 `assert.equal(true, !!value, message);` 。

assert.equal(actual, expected[, message])

判等 `actual` 与 `expected` 是否相等，等同于使用 `==` 进行比较。

assert.notEqual(actual, expected[, message])

判断 `actual` 与 `expected` 是否不相等，等同于使用 `!=` 进行比较。

assert.deepEqual(actual, expected[, message])

深度判断相等，通过比较 `actual` 与 `expected` 所有原型（prototype）之外的属性是否相等（ `==`)来判断二者是否相等。

assert.notDeepEqual(actual, expected[, message])

深度判断不相等，与 `assert.deepEqual` 的结果相反。

assert.strictEqual(actual, expected[, message])

判断 `actual` 与 `expected` 是否“全等（ `===` ）”。

assert.notStrictEqual(actual, expected[, message])

判断 `actual` 与 `expected` 是否“不全等（ `!==` ）”。

assert.deepStrictEqual(actual, expected[, message])

深度判断全等，通过比较 `actual` 与 `expected` 所有原型（prototype）之外的属性是否全等（ `===` ）来判断二者是否相等。

assert.notDeepStrictEqual(actual, expected[, message])

深度判断不全等，与 `assert.deepStrictEqual` 结果相反。

assert.throws(block[, error][, message])

期望 `block` 抛出一个 `error` 。 `error` 可以是构造函数，正则表达式，或验证函数。

使用构造函数验证实例：

```
assert.throws(
  function() {
    throw new Error("Wrong value");
  },
  Error
);
```

使用正则表达式验证错误信息:

```
assert.throws(
  function() {
    throw new Error("Wrong value");
  },
  /value/
);
```

自定义错误验证:

```
assert.throws(
  function() {
    throw new Error("Wrong value");
  },
  function(err) {
    if ( (err instanceof Error) && /value/.test(err) ) {
      return true;
    }
  },
  "unexpected error"
);
```

assert.doesNotThrow(block[, message])

期望 `block` 不抛出错误，详情见 `assert.throws` 。

assert.ifError(value)

测试 `value` 是否为假，当 `value` 为真时会抛出异常。通常用来判断回调函数中第一个 `error` 参数。

Buffer

稳定度: 2 - 稳定

纯粹的 `JavaScript` 是Unicode友好的，但是不能很好地处理二进制数据。当处理TCP流或者文件流时，操作八进制流是必要的。 `node.js` 提供了多种策略来操作，创建和使用八进制流。

原始的数据被存储在 `Buffer` 类的实例中，一个 `Buffer` 类似于一个整数数组但是使用了V8堆之外的内存分配。一个 `Buffer` 不能被改变大小。

`Buffer` 类是全局的，所以它是少数的不用 `require('buffer')` 就能使用的对象之一。

`Buffer` 和 `JavaScript` 字符串对象之间的转换需要指定一个明确地编码方法。以下是一些不同的字符串编码。

'ascii' - 仅供7位的ASCII数据使用，这个编码方法非常的快速，而且会剥离过高的位（如果有设置）。

'utf8' - 多字节编码的字符。许多web页面和一些其他文档都使用UTF-8编码。

'utf16le' - 2或4个字节， `little endian` 编码字符。支持(U+10000 到 U+10FFFF)的代理对。

'ucs2' - 'utf16le'的别名。

'base64' - Base64 字符串编码。

'binary' - 一种通过使用每个字符的前八位来将二进制数据解码为字符串的方式。这个编码方法已经不被推荐使用，在处理 `Buffer` 对象时应避免使用它，这个编码将会在 `node.js` 的未来版本中移除。

'hex' - 把每个字节编码成2个十六进制字符。

从一个 `Buffer` 创建一个类型数组(typed array)遵循以下的说明：

1， `Buffer` 的内存是被复制的，不是共享的。

2，`Buffer`的内存被解释当做一个数组，而不是一个字节数组(byte array)。换言之，`new Uint32Array(new Buffer([1,2,3,4]))`创建了一个4个元素（[1,2,3,4]）的`Uint32Array`,不是一个只有一个元素（[0x1020304] 或 [0x4030201]）的`Uint32Array`。

注意：Node.js v0.8 只是简单得在`array.buffer`中保留了`buffer`的引用，而不是复制一份。

Class: Buffer

Buffer类是一个全局类用于直接处理二进制数据。它的实例可以被多种途径构建。

new Buffer(size)

- size Number

分配一个大小为指定`size`的八位字节的新`buffer`。注意，`size`不能超过kMaxLength，否则一个`RangeError`将会被抛出。

new Buffer(array)

- array Array

使用一个八进制数组分配一个新的`buffer`。

new Buffer(buffer)

- buffer Buffer

将传递的`buffer`复制进一个新的Buffer实例。

new Buffer(str[, encoding])

- str String - 传入buffer的字符串
- encoding String - 可选，使用的编码

根据给定的`str`创建一个新的`buffer`，编码默认是UTF-8。

Class Method: Buffer.isEncoding(encoding)

- encoding String 将要被测试的编码，返回`true`若此编码是合法的，否则返

回 `false` 。

Class Method: Buffer.isBuffer(obj)

- obj Object
- Return: Boolean

测试 `obj` 是否为一个 `buffer` 。

Class Method: Buffer.byteLength(string[, encoding])

- string String
- encoding String, 可选， 默认： 'utf8'
- Return: Number

给出 `string` 的实际字节长度。编码默认为UTF-8。这与 `String.prototype.length` 不同，因为 `String.prototype.length` 只返回字符串中字符的数量。

例子:

```
str = '\u00bd + \u00bc = \u00be';

console.log(str + ": " + str.length + " characters, " +
  Buffer.byteLength(str, 'utf8') + " bytes");

// ½ + ¼ = ¾: 9 characters, 12 bytes
```

Class Method: Buffer.concat(list[, totalLength])

- list Array 需要被连接的Buffer对象
- totalLength Number 将要被连接的buffers的
- Returns 连接完毕的buffer

若 `list` 为空，或 `totalLength` 为0，那么将返回一个长度为0的buffer。若 `list` 只有一个元素，那么这个元素将被返回。 若 `list` 有超过一个元素，那么将创建一个新的Buffer实例。 如果 `totalLength` 没有被提供，那么将会从 `list` 中计算读取。但是，这增加了函数的一个额外的循环，所以如果直接长度那么性能会更好。

Class Method: Buffer.compare(buf1, buf2)

- buf1 Buffer
- buf2 Buffer 与buf1.compare(buf2)相同. 对于排序一个Buffers的数组非常有用:

  ```
  var arr = [Buffer('1234'), Buffer('0123')];
  arr.sort(Buffer.compare);
  ```

buf.length

- Return Number 这个buffer的字节长度。注意这不一定是这个buffer中的内容长度。它是这个buffer对象所分配内存大小，并不会随着buffer的内容的改变而改变

```
buf = new Buffer(1234);

console.log(buf.length);
buf.write("some string", 0, "ascii");
console.log(buf.length);

// 1234
// 1234
```

虽然 `buffer` 的 `length` 属性并不是不可变的，改变 `length` 属性的值可能会使之变成 `undefined` 或引起一些不一致的行为。希望去改变 `buffer` 的 `length` 的应用应当把它视作一个只读的值，并且使用 `buf.slice` 来创建一个新的 `buffer` 。

```
buf = new Buffer(10);
buf.write("abcdefghj", 0, "ascii");
console.log(buf.length); // 10
buf = buf.slice(0,5);
console.log(buf.length); // 5
```

buf.write(string[, offset][, length][, encoding])

- string String 准备被写入buffer的数据
- offset Number 可选，默认为0

- length Number 可选，默认为 `buffer.length - offset`
- encoding String 可选，默认为 'utf8'

从指定的偏移位置(offset)使用给定的编码向buffer中写入字符串，偏移位置默认为0，编码默认为UTF8。长度为将要写入的字符串的字节大小。返回被写入的八进制流的大小。如果buffer没有足够的空间写入整个字符串，那么它将只会写入一部分。 `length` 参数默认为 `buffer.length - offset` ，这个方法将不会只写入字符的一部分。

```
buf = new Buffer(256);
len = buf.write('\u00bd + \u00bc = \u00be', 0);
console.log(len + " bytes: " + buf.toString('utf8', 0, len));
```

buf.writeUIntLE(value, offset, byteLength[, noAssert])

buf.writeUIntBE(value, offset, byteLength[, noAssert])

buf.writeIntLE(value, offset, byteLength[, noAssert])

buf.writeIntBE(value, offset, byteLength[, noAssert])

- value {Number} 将要被写入buffer的字节
- offset {Number} `0 <= offset <= buf.length`
- byteLength {Number} `0 < byteLength <= 6`
- noAssert {Boolean} 默认为 `false`
- Return: {Number}

根据指定的偏移位置(offset)和 `byteLength` 将 `value` 写入buffer。最高支持48位的精确度。例子：

```
var b = new Buffer(6);
b.writeUIntBE(0x1234567890ab, 0, 6);
// <Buffer 12 34 56 78 90 ab>
```

将 `noAssert` 设置为true将会跳过 `value` 和 `offset` 的检验，默认为 `false` 。

buf.readUIntLE(offset, byteLength[, noAssert])

buf.readUIntBE(offset, byteLength[, noAssert])

buf.readIntLE(offset, byteLength[, noAssert])

buf.readIntBE(offset, byteLength[, noAssert])

- offset {Number} `0 <= offset <= buf.length`
- byteLength {Number} `0 < byteLength <= 6`
- noAssert {Boolean} 默认为 `false`
- Return: {Number}

一个普遍的用来作数值读取的方法，最高支持48位的精确度。例子：

```
var b = new Buffer(6);
b.writeUint16LE(0x90ab, 0);
b.writeUInt32LE(0x12345678, 2);
b.readUIntLE(0, 6).toString(16);  // Specify 6 bytes (48 bits)
// output: '1234567890ab'
```

将 `noAssert` 设置为true将会跳过 `value` 和 `offset` 的检验，这意味着 `offset` 将可能超过buffer的结束位置，默认为 `false` 。

buf.toString([encoding][, start][, end])

- encoding String, 可选，默认为 'utf8'
- start Number, 可选，默认为 0
- end Number, 可选默认为 `buffer.length`

从编码的buffer数据中使用指定的编码解码并返回结果字符串。如果 `encoding` 为 `undefined` 或 `null` ，那么 `encoding` 将默认为UTF8。 `start` 和 `end` 参数默认为 `0` 和 `buffer.length` 。

```
buf = new Buffer(26);
for (var i = 0 ; i < 26 ; i++) {
  buf[i] = i + 97; // 97 is ASCII a
}
buf.toString('ascii'); // outputs: abcdefghijklmnopqrstuvwxyz
buf.toString('ascii',0,5); // outputs: abcde
buf.toString('utf8',0,5); // outputs: abcde
buf.toString(undefined,0,5); // encoding defaults to 'utf8', outputs abcde
```

buf.toJSON()

返回一个Buffer实例的JSON形式。 `JSON.stringify` 被隐式得调用当转换Buffer实例时。

例子:

```
var buf = new Buffer('test');
var json = JSON.stringify(buf);

console.log(json);
// '{"type":"Buffer","data":[116,101,115,116]}'

var copy = JSON.parse(json, function(key, value) {
    return value && value.type === 'Buffer'
      ? new Buffer(value.data)
      : value;
  });

console.log(copy);
// <Buffer 74 65 73 74>
```

buf[index]

获取或设置指定位置的八位字节。这个值是指单个字节，所有合法范围在 `0x00` 到 `0xFF` 或 `0` 到 `255` 。

例子：复制一个ASCII字符串到一个buffer，一次一个字节：

```
str = "node.js";
buf = new Buffer(str.length);

for (var i = 0; i < str.length ; i++) {
  buf[i] = str.charCodeAt(i);
}

console.log(buf);

// node.js
```

buf.equals(otherBuffer)

- otherBuffer Buffer

返回一个布尔值表示是否 `buf` 与 `otherBuffer` 具有相同的字节。

buf.compare(otherBuffer)

- otherBuffer Buffer

返回一个数字表示在排序上 `buf` 在 `otherBuffer` 之前，之后或相同。

buf.copy(targetBuffer[, targetStart][, sourceStart][, sourceEnd])

- targetBuffer Buffer 将要进行复制的Buffer
- targetStart Number 可选，默认为 0
- sourceStart Number 可选，默认为 0
- sourceEnd Number 可选，默认为 `buffer.length`

从 `buf` 中的指定范围复制数据到 `targetBuffer` 中的指定范围，它们是可以重叠的。

例子：创建两个Buffer，然后复制buf1的第16字节到19字节到buf2，buf2的偏移位置从第8字节开始：

```
buf1 = new Buffer(26);
buf2 = new Buffer(26);

for (var i = 0 ; i < 26 ; i++) {
  buf1[i] = i + 97; // 97 is ASCII a
  buf2[i] = 33; // ASCII !
}

buf1.copy(buf2, 8, 16, 20);
console.log(buf2.toString('ascii', 0, 25));

// !!!!!!!!qrst!!!!!!!!!!!!!
```

例子：创建一个单独的Buffer，然后复制数据到自身的一个重叠的范围。

```
buf = new Buffer(26);

for (var i = 0 ; i < 26 ; i++) {
  buf[i] = i + 97; // 97 is ASCII a
}

buf.copy(buf, 0, 4, 10);
console.log(buf.toString());

// efghijghijklmnopqrstuvwxyz
```

buf.slice([start][, end])

- start Number 可选，默认为 0
- end Number 可选，默认为 `buffer.length`
- 返回一个和旧的buffer引用了相同内存的新的buffer，但是被 `start` 和 `end` 参数所偏移和裁剪。

修改这个新的buffer的切片，也会改变内存中原来的buffer。

例子：创建一个ASCII字母的Buffer，然后对其进行 `slice` ，然后修改源Buffer上的一个字节：

```
var buf1 = new Buffer(26);

for (var i = 0 ; i < 26 ; i++) {
  buf1[i] = i + 97; // 97 is ASCII a
}

var buf2 = buf1.slice(0, 3);
console.log(buf2.toString('ascii', 0, buf2.length));
buf1[0] = 33;
console.log(buf2.toString('ascii', 0, buf2.length));

// abc
// !bc
```

buf.indexOf(value[, byteOffset])

- value String Buffer或Number
- byteOffset Number 可选，默认为 0
- Return: Number

行为和Array.indexOf()相似。接受一个字符串，Buffer或数字。字符串被解释为UTF8编码，Buffer将使用整个buffer，所以如果要比较部分的Buffer请使用 `Buffer.slice()` ，数字的范围需在0到255之间。

buf.readUInt8(offset[, noAssert])

- offset Number
- noAssert Boolean 可选，默认为 `false`
- Return: Number

根据制定偏移量从buffer中读取一个无符号8位整数。

将 `noAssert` 设置为true将会跳过 `value` 和 `offset` 的检验，这意味着 `offset` 将可能超过buffer的结束位置，默认为 `false` 。

例子:

```
var buf = new Buffer(4);

buf[0] = 0x3;
buf[1] = 0x4;
buf[2] = 0x23;
buf[3] = 0x42;

for (ii = 0; ii < buf.length; ii++) {
  console.log(buf.readUInt8(ii));
}

// 0x3
// 0x4
// 0x23
// 0x42
```

buf.readUInt16LE(offset[, noAssert])

buf.readUInt16BE(offset[, noAssert])

- offset Number
- noAssert Boolean 可选，默认为 `false`
- Return: Number

根据制定偏移量从buffer中根据特定的 `endian` 字节序读取一个无符号16位整数。

将 `noAssert` 设置为true将会跳过 `value` 和 `offset` 的检验，这意味着 `offset` 将可能超过buffer的结束位置，默认为 `false` 。

例子:

```
var buf = new Buffer(4);

buf[0] = 0x3;
buf[1] = 0x4;
buf[2] = 0x23;
buf[3] = 0x42;

console.log(buf.readUInt16BE(0));
console.log(buf.readUInt16LE(0));
console.log(buf.readUInt16BE(1));
console.log(buf.readUInt16LE(1));
console.log(buf.readUInt16BE(2));
console.log(buf.readUInt16LE(2));

// 0x0304
// 0x0403
// 0x0423
// 0x2304
// 0x2342
// 0x4223
```

buf.readUInt32LE(offset[, noAssert])

buf.readUInt32BE(offset[, noAssert])

- offset Number
- noAssert Boolean 可选，默认为 `false`
- Return: Number

根据制定偏移量从buffer中根据特定的 `endian` 字节序读取一个无符号32位整数。

将 `noAssert` 设置为true将会跳过 `value` 和 `offset` 的检验，这意味着 `offset` 将可能超过buffer的结束位置，默认为 `false` 。

例子：

```
var buf = new Buffer(4);

buf[0] = 0x3;
buf[1] = 0x4;
buf[2] = 0x23;
buf[3] = 0x42;

console.log(buf.readUInt32BE(0));
console.log(buf.readUInt32LE(0));

// 0x03042342
// 0x42230403
```

buf.readInt8(offset[, noAssert])

- offset Number
- noAssert Boolean 可选，默认为 `false`
- Return: Number

根据制定偏移量从buffer中读取一个有符号8位整数。

将 `noAssert` 设置为true将会跳过 `value` 和 `offset` 的检验，这意味着 `offset` 将可能超过buffer的结束位置，默认为 `false` 。

运作和 `buffer.readUInt8` 相同，除非buffer内容中有包含了作为2的补码的有符号值。

buf.readInt16LE(offset[, noAssert])

buf.readInt16BE(offset[, noAssert])

- offset Number
- noAssert Boolean 可选，默认为 `false`
- Return: Number

根据制定偏移量从buffer中根据特定的 `endian` 字节序读取一个有符号16位整数。

将 `noAssert` 设置为true将会跳过 `value` 和 `offset` 的检验，这意味着 `offset` 将可能超过buffer的结束位置，默认为 `false` 。

运作和 `buffer.readUInt16` 相同，除非buffer内容中有包含了作为2的补码的有符号值。

buf.readInt32LE(offset[, noAssert])

buf.readInt32BE(offset[, noAssert])

- offset Number
- noAssert Boolean 可选，默认为 `false`
- Return: Number

根据制定偏移量从buffer中根据特定的 `endian` 字节序读取一个有符号32位整数。

将 `noAssert` 设置为true将会跳过 `value` 和 `offset` 的检验，这意味着 `offset` 将可能超过buffer的结束位置，默认为 `false` 。

运作和 `buffer.readUInt32` 相同，除非buffer内容中有包含了作为2的补码的有符号值。

buf.readFloatLE(offset[, noAssert])

buf.readFloatBE(offset[, noAssert])

- offset Number
- noAssert Boolean 可选，默认为 `false`
- Return: Number

根据制定偏移量从buffer中根据特定的 `endian` 字节序读取一个32位浮点数。

将 `noAssert` 设置为true将会跳过 `value` 和 `offset` 的检验，这意味着 `offset` 将可能超过buffer的结束位置，默认为 `false` 。

例子：

```
var buf = new Buffer(4);

buf[0] = 0x00;
buf[1] = 0x00;
buf[2] = 0x80;
buf[3] = 0x3f;

console.log(buf.readFloatLE(0));

// 0x01
```

buf.readDoubleLE(offset[, noAssert])

buf.readDoubleBE(offset[, noAssert])

- offset Number
- noAssert Boolean 可选，默认为 `false`
- Return: Number

根据制定偏移量从buffer中根据特定的 `endian` 字节序读取一个64位双精度数。

将 `noAssert` 设置为true将会跳过 `value` 和 `offset` 的检验，这意味着 `offset` 将可能超过buffer的结束位置，默认为 `false` 。

例子：

```
var buf = new Buffer(8);

buf[0] = 0x55;
buf[1] = 0x55;
buf[2] = 0x55;
buf[3] = 0x55;
buf[4] = 0x55;
buf[5] = 0x55;
buf[6] = 0xd5;
buf[7] = 0x3f;

console.log(buf.readDoubleLE(0));

// 0.3333333333333333
```

buf.writeUInt8(value, offset[, noAssert])

- value Number
- offset Number
- noAssert Boolean 可选，默认为 `false`

向 `buffer` 的指定偏移位置写入 `value` 。 注意， `value` 必须是一个合法的无符号8位整形数。

将 `noAssert` 设置为 `true` 将跳过 `value` 和 `offset` 的验证。这意味着 `value` 可能会过大，或者 `offset` 超过 `buffer` 的末尾导致 `value` 被丢弃，这个参数除非你十分有把握否则你不应去使用它，默认为 `false` 。

例子：

```
var buf = new Buffer(4);
buf.writeUInt8(0x3, 0);
buf.writeUInt8(0x4, 1);
buf.writeUInt8(0x23, 2);
buf.writeUInt8(0x42, 3);

console.log(buf);

// <Buffer 03 04 23 42>
```

buf.writeUInt16LE(value, offset[, noAssert])

buf.writeUInt16BE(value, offset[, noAssert])

- value Number
- offset Number
- noAssert Boolean 可选，默认为 `false`

向 `buffer` 的指定偏移位置根据特定的 `endian` 字节序写入 `value` 。注意， `value` 必须是一个合法的无符号16位整形数。

将 `noAssert` 设置为 `true` 将跳过 `value` 和 `offset` 的验证。这意味着 `value` 可能会过大，或者 `offset` 超过 `buffer` 的末尾导致 `value` 被丢弃，这个参数除非你十分有把握否则你不应去使用它，默认为 `false` 。

例子：

```
var buf = new Buffer(4);
buf.writeUInt16BE(0xdead, 0);
buf.writeUInt16BE(0xbeef, 2);

console.log(buf);

buf.writeUInt16LE(0xdead, 0);
buf.writeUInt16LE(0xbeef, 2);

console.log(buf);

// <Buffer de ad be ef>
// <Buffer ad de ef be>
```

buf.writeUInt32LE(value, offset[, noAssert])

buf.writeUInt32BE(value, offset[, noAssert])

- value Number
- offset Number
- noAssert Boolean 可选，默认为 `false`

向 `buffer` 的指定偏移位置根据特定的 `endian` 字节序写入 `value` 。 注意， `value` 必须是一个合法的无符号32位整形数。

将 `noAssert` 设置为 `true` 将跳过 `value` 和 `offset` 的验证。这意味着 `value` 可能会过大，或者 `offset` 超过 `buffer` 的末尾导致 `value` 被丢弃，这个参数除非你十分有把握否则你不应去使用它，默认为 `false` 。

例子：

```
var buf = new Buffer(4);
buf.writeUInt32BE(0xfeedface, 0);

console.log(buf);

buf.writeUInt32LE(0xfeedface, 0);

console.log(buf);

// <Buffer fe ed fa ce>
// <Buffer ce fa ed fe>
```

buf.writeInt8(value, offset[, noAssert])

- value Number
- offset Number
- noAssert Boolean 可选，默认为 `false`

向 `buffer` 的指定偏移位置中写入 `value` 。 注意， `value` 必须是一个合法的无符号32位整形数。

将 `noAssert` 设置为 `true` 将跳过 `value` 和 `offset` 的验证。这意味着 `value` 可能会过大，或者 `offset` 超过 `buffer` 的末尾导致 `value` 被丢弃，这个参数除非你十分有把握否则你不应去使用它，默认为 `false` 。

运作和 `buffer.writeUInt8` 相同，除非buffer内容中有包含了作为2的补码的有符号值。

buf.writeInt16LE(value, offset[, noAssert])

buf.writeInt16BE(value, offset[, noAssert])

- value Number
- offset Number
- noAssert Boolean 可选，默认为 `false`

向 `buffer` 的指定偏移位置根据特定的 `endian` 字节序写入 `value` 。 注意， `value` 必须是一个合法的有符号16位整形数。

将 `noAssert` 设置为 `true` 将跳过 `value` 和 `offset` 的验证。这意味着 `value` 可能会过大，或者 `offset` 超过 `buffer` 的末尾导致 `value` 被丢弃，这个参数除非你十分有把握否则你不应去使用它，默认为 `false` 。

运作和 `buffer.writeUInt16` 相同，除非buffer内容中有包含了作为2的补码的有符号值。

buf.writeInt32LE(value, offset[, noAssert])

buf.writeInt32BE(value, offset[, noAssert])

- value Number
- offset Number
- noAssert Boolean 可选，默认为 `false`

向 `buffer` 的指定偏移位置根据特定的 `endian` 字节序写入 `value` 。注意， `value` 必须是一个合法的有符号32位整形数。

将 `noAssert` 设置为 `true` 将跳过 `value` 和 `offset` 的验证。这意味着 `value` 可能会过大，或者 `offset` 超过 `buffer` 的末尾导致 `value` 被丢弃，这个参数除非你十分有把握否则你不应去使用它，默认为 `false` 。

运作和 `buffer.writeUInt32` 相同，除非buffer内容中有包含了作为2的补码的有符号值。

buf.writeFloatLE(value, offset[, noAssert])

buf.writeFloatBE(value, offset[, noAssert])

- value Number
- offset Number
- noAssert Boolean 可选，默认为 `false`

向 `buffer` 的指定偏移位置根据特定的 `endian` 字节序写入 `value` 。注意， `value` 必须是一个合法的32位浮点数。

将 `noAssert` 设置为 `true` 将跳过 `value` 和 `offset` 的验证。这意味着 `value` 可能会过大，或者 `offset` 超过 `buffer` 的末尾导致 `value` 被丢弃，这个参数除非你十分有把握否则你不应去使用它，默认为 `false` 。

例子：

```
var buf = new Buffer(4);
buf.writeFloatBE(0xcafebabe, 0);

console.log(buf);

buf.writeFloatLE(0xcafebabe, 0);

console.log(buf);

// <Buffer 4f 4a fe bb>
// <Buffer bb fe 4a 4f>
```

buf.writeDoubleLE(value, offset[, noAssert])

buf.writeDoubleBE(value, offset[, noAssert])

- value Number
- offset Number
- noAssert Boolean 可选，默认为 `false`

向 `buffer` 的指定偏移位置根据特定的 `endian` 字节序写入 `value` 。注意， `value` 必须是一个合法的64位双精度数。

将 `noAssert` 设置为 `true` 将跳过 `value` 和 `offset` 的验证。这意味着 `value` 可能会过大，或者 `offset` 超过 `buffer` 的末尾导致 `value` 被丢弃，这个参数除非你十分有把握否则你不应去使用它，默认为 `false` 。

例子：

```
var buf = new Buffer(8);
buf.writeDoubleBE(0xdeadbeefcafebabe, 0);

console.log(buf);

buf.writeDoubleLE(0xdeadbeefcafebabe, 0);

console.log(buf);

// <Buffer 43 eb d5 b7 dd f9 5f d7>
// <Buffer d7 5f f9 dd b7 d5 eb 43>
```

buf.fill(value[, offset][, end])

- value
- offset Number 可选
- end Number 可选

使用指定的 `value` 填充buffer。如果 `offset` （默认为0）和 `end` （默认为 `buffer.length` ）没有指定，将会填充整个buffer。

```
var b = new Buffer(50);
b.fill("h");
```

buffer.values()

创建一个buffer内的值（bytes）的迭代器。这个函数会被自动调用当buffer被用于 `for..of` 语句中时。

buffer.keys()

创建一个buffer的索引的迭代器。

buffer.entries()

创建一个[index, byte]数组迭代器。

buffer.INSPECT_MAX_BYTES

Number 默认值： 50

表示有多少字节会被返回当调用 `buffer.inspect()` 时。它可以被用户的模块所覆盖。

注意这是一个由 `require('buffer')` 返回的 `buffer模块` 的属性，并不是全局 `Buffer` 对象或buffer实例的。

ES6 迭代器

`Buffers` 可以被ES6的 `for..of` 语法迭代：

```
var buf = new Buffer([1, 2, 3]);

for (var b of buf)
  console.log(b)

// 1
// 2
// 3
```

另外的， `buffer.values()` ， `buffer.keys()` 和 `buffer.entries()` 方法都可以被用来创建迭代器。

Class: SlowBuffer

返回一个不被池管理的 `Buffer` 。

为了避免创建许多单个的被分配内存的小Buffer的垃圾回收开销。默认得，分配小于4KB的空间将会被从一个更大的被分配好内存的对象（allocated object）中切片(sliced)得到。这个方法改进了性能以及内存占用，因为V8的垃圾回收机制不再需要追踪和清理许多的小对象。

当开发者需要将池中一小块数据保留不确定的一段时间，较为妥当的办法是用 `SlowBuffer` 创建一个不被池管理的 `Buffer` 实例并将相应数据拷贝出来。

```
// need to keep around a few small chunks of memory
var store = [];

socket.on('readable', function() {
  var data = socket.read();
  // allocate for retained data
  var sb = new SlowBuffer(10);
  // copy the data into the new allocation
  data.copy(sb, 0, 0, 10);
  store.push(sb);
});
```

请谨慎使用，仅作为开发者察觉到在应用中有过度的内存保留时的最后手段。

Child Process

稳定度: 2 - 稳定

`node.js` 通过 `child_process` 模块提供了三向的 `popen` 功能。

可以无阻塞地通过子进程的 `stdin` ， `stdout` 和 `stderr` 以流的方式传递数据。（注意某些程序在内部使用了行缓冲I/O，这不会影响 `node.js` ，但是这意味你传递给子进程的数据可能不会在第一时间被消费）。

可以通过 `require('child_process').spawn()` 或 `require('child_process').fork()` 创建子进程。这两者间的语义有少许差别，将会在后面进行解释。

当以写脚本为目的时，你可以会觉得使用同步版本的方法会更方便。

Class: ChildProcess

`ChildProcess` 是一个 `EventEmitter` 。

子进程总是有三个与之相关的流。 `child.stdin` ， `child.stdout` 和 `child.stderr` 。他们可能会共享父进程的stdio流，或者也可以是独立的被导流的流对象。

`ChildProcess` 类并不是用来直接被使用的。应当使用 `spawn()` , `exec()` , `execFile()` 或 `fork()` 方法来创建一个子进程实例。

Event: 'error'

- err Error 错误对象

发生于：

进程不能被创建时，进程不能杀死时，给子进程发送信息失败时。 注意 `exit` 事件在一个错误发生后可能触发。如果你同时监听了这两个事件来触发一个函数，需要记住不要让这个函数被触发两次。

参阅 `ChildProcess.kill()` 和 `ChildProcess.send()` 。

Event: 'exit'

- code Number 如果进程正常退出，则为退出码。如果进程被父进程杀死，则为被传递的信号字符串。这个事件将在子进程结束运行时被触发。

注意子进程的stdio流可能仍为打开状态。

还需要注意的是， `node.js` 已经为我们添加了'SIGINT'信号和'SIGTERM'信号的事件处理函数，所以在父进程发出这两个信号时，进程将会退出。

参阅 `waitpid(2)` 。

Event: 'close'

- code Number 如果进程正常退出，则为退出码。如果进程被父进程杀死，则为被传递的信号字符串。这个事件将在子进程结束运行时被触发。这个事件将会在子进程的 `stdio` 流都关闭时触发。这是与 `exit` 的区别，因为可能会有几个进程共享同样的 `stdio` 流。

Event: 'disconnect'

在父进程或子进程中使用 `.disconnect()` 方法后这个事件会触发。在断开之后，将不能继续相互发送信息，并且子进程的 `.connected` 属性将会是 `false` 。

Event: 'message'

- message Object 一个已解析的JSON对象或一个原始类型值
- sendHandle Handle object 一个 `Socket` 或 `Server` 对象

通过 `.send(message, [sendHandle])` 发送的信息可以通过监听 `message` 事件获取到。

child.stdin

- Stream object

一个代表了子进程的 `stdin` 的可写流。通过 `end()` 方法关闭此流可以终止子进程。

如果子进程通过 `spawn` 创建时 `stdio` 没有被设置为 `pipe` ，那么它将不会被创建。

`child.stdin` 为 `child.stdio` 中对应元素的快捷引用。它们要么都指向同一个对象，要么都为null。

child.stdout

- Stream object

一个代表了子进程的 `stdout` 的可读流。

如果子进程通过 `spawn` 创建时 `stdio` 没有被设置为 `pipe` ，那么它将不会被创建。

`child.stdout` 为 `child.stdio` 中对应元素的快捷引用。它们要么都指向同一个对象，要么都为null。

child.stderr

- Stream object

一个代表了子进程的 `stderr` 的可读流。

如果子进程通过 `spawn` 创建时 `stdio` 没有被设置为 `pipe` ，那么它将不会被创建。

`child.stderr` 为 `child.stdio` 中对应元素的快捷引用。它们要么都指向同一个对象，要么都为null。

child.stdio

- Array

一个包含了子进程的管道的稀疏数组，元素的位置对应着利用 `spawn` 创建子进程时 `stdio` 配置参数里被设置为 `pipe` 的位置。注意索引为0-2的流分别与 `ChildProcess.stdin` ， `ChildProcess.stdout` 和 `ChildProcess.stderr` 引用的是相同的对象。

在下面的例子中，在 `stdio` 参数中只有索引为1的元素被设置为了 `pipe` ，所以父进程中只有 `child.stdio[1]` 是一个流，其他的元素都为 `null` 。

```
var assert = require('assert');
var fs = require('fs');
var child_process = require('child_process');

child = child_process.spawn('ls', {
    stdio: [
      0, // use parents stdin for child
      'pipe', // pipe child's stdout to parent
      fs.openSync('err.out', 'w') // direct child's stderr to a
file
    ]
});

assert.equal(child.stdio[0], null);
assert.equal(child.stdio[0], child.stdin);

assert(child.stdout);
assert.equal(child.stdio[1], child.stdout);

assert.equal(child.stdio[2], null);
assert.equal(child.stdio[2], child.stderr);
```

child.pid

- Integer

子进程的 `PID` 。

例子：

```
var spawn = require('child_process').spawn,
    grep  = spawn('grep', ['ssh']);

console.log('Spawned child pid: ' + grep.pid);
grep.stdin.end();
```

child.connected

- Boolean 在 `.disconnect` 方法被调用后将会被设置为 `false` 。如

果 `.connected` 属性为 `false` ，那么将不能再向子进程发送信息。

child.kill([signal])

- signal String

给子进程传递一个信号。如果没有指定任何参数，那么将发送 `'SIGTERM'` 给子进程。更多可用的信号请参阅 `signal(7)` 。

```
var spawn = require('child_process').spawn,
    grep  = spawn('grep', ['ssh']);

grep.on('close', function (code, signal) {
  console.log('child process terminated due to receipt of signal
 ' + signal);
});

// send SIGHUP to process
grep.kill('SIGHUP');
```

在信号不能被送达时，可能会产生一个 `error` 事件。给一个已经终止的子进程发送一个信号不会发生错误，但可以操作不可预料的后果：如果该子进程的 `PID` 已经被重新分配给了另一个进程，那么这个信号会被传递到另一个进程中。大家可以猜想这将会发生什么样的情况。

注意这个函数仅仅是名字叫kill，给子进程发送的信号可能不是去关闭它的。这个函数仅仅只是给子进程发送一个信号。

参阅 `kill(2)` 。

child.send(message[, sendHandle])

- message Object
- sendHandle Handle object

当使用 `child_process.fork()` 时，你可以使用 `child.send(message, [sendHandle])` 向子进程发送信息，子进程里会触发 `message` 事件当收到信息时。

例子：

```
var cp = require('child_process');

var n = cp.fork(__dirname + '/sub.js');

n.on('message', function(m) {
  console.log('PARENT got message:', m);
});

n.send({ hello: 'world' });
```

子进程代码, `sub.js` 可能看起来类似这样:

```
process.on('message', function(m) {
  console.log('CHILD got message:', m);
});

process.send({ foo: 'bar' });
```

在子进程中，`process` 对象将有一个 `send()` 方法，在它的信道上收到一个信息时，信息将以对象的形式返回。

请注意父进程，子进程中的 `send()` 方法都是同步的，所以发送大量数据是不被建议的（可以使用管道代替，参阅 `child_process.spawn` ）。

发送 `{cmd: 'NODE_foo'}` 信息时是一个特殊情况。所有的在 `cmd` 属性中包含了 `NODE_` 前缀的信息都不会触发 `message` 事件，因为这是 `node.js` 内核使用的内部信息。包含这个前缀的信息都会触发 `internalMessage` 事件。请避免使用这个事件，它在改变的时候不会收到通知。

`child.send()` 的 `sendHandle` 参数时用来给另一个进程发送一个 `TCP服务器` 或一个 `socket` 的。将之作为第二个参数传入，子进程将在 `message` 事件中会收到这个对象。

如果信息不能被发送的话将会触发一个 `error` 事件，比如子进程已经退出了。

例子：发送一个 `server` 对象

```
var child = require('child_process').fork('child.js');

// Open up the server object and send the handle.
var server = require('net').createServer();
server.on('connection', function (socket) {
  socket.end('handled by parent');
});
server.listen(1337, function() {
  child.send('server', server);
});
```

子进程将会收到 `server` 对象：

```
process.on('message', function(m, server) {
  if (m === 'server') {
    server.on('connection', function (socket) {
      socket.end('handled by child');
    });
  }
});
```

注意这个 `server` 现在已经被父进程和子进程所共享，这意味着链接将可能被父进程处理也可能被子进程处理。

对于 `dgram` 服务器，流程也是完全一样的。使用 `message` 事件而不是 `connection` 事件，使用 `server.bind` 问不是 `server.listen` （目前只支持 `UNIX` 平台）。

例子：发送一个 `socket` 对象

以下是发送一个 `socket` 的例子。创建了两个子进程。并且将地址为 `74.125.127.100` 的链接通过将 `socket` 发送给"special"子进程来视作VIP。其他的 `socket` 则被发送给"normal"子进程。

```
var normal = require('child_process').fork('child.js', ['normal'
]);
var special = require('child_process').fork('child.js', ['specia
l']);

// Open up the server and send sockets to child
var server = require('net').createServer();
server.on('connection', function (socket) {

  // if this is a VIP
  if (socket.remoteAddress === '74.125.127.100') {
    special.send('socket', socket);
    return;
  }
  // just the usual dudes
  normal.send('socket', socket);
});
server.listen(1337);

`child.js`:
```js
process.on('message', function(m, socket) {
  if (m === 'socket') {
    socket.end('You were handled as a ' + process.argv[2] + ' pe
rson');
  }
});
```

注意一旦一个单独的 `socket` 被发送给了子进程，那么父进程将不能追踪到这个 `socket` 被删除的时间，这个情况下 `.connections` 属性将会成为 `null` 。在这个情况下同样也不推荐使用 `.maxConnections` 属性。

child.disconnect()

关闭父进程与子进程间的IPC信道，它让子进程非常优雅地退出，因为已经活跃的信道了。在调用了这个方法后，父进程和子进程的 `.connected` 标签都会被设置为 `false` ，将不能再发送信息。

`disconnect` 事件在进程不再有消息接收时触发。
```

注意，当子进程中有与父进程通信的IPC信道时，你也可以在子进程中调用 `process.disconnect()` 。

异步进程的创建

以下方法遵循普遍的异步编程模式（接受一个回调函数或返回一个 `EventEmitter` ）。

child_process.spawn(command[, args][, options])

- command String 将要运行的命令
- args Array 字符串参数数组
- **options Object**
  - cwd String 子进程的当前工作目录
  - env Object 环境变量键值对
  - stdio Array|String 子进程的stdio配置
  - detached Boolean 这个子进程将会变成进程组的领导
  - uid Number 设置用户进程的ID
  - gid Number 设置进程组的ID
- return: ChildProcess object

利用给定的命令以及参数执行一个新的进程，如果没有参数数组，那么 `args` 将默认是一个空数组。

第三个参数时用来指定以为额外的配置，以下是它的默认值：

```
{ cwd: undefined,
  env: process.env
}
```

使用 `cwd` 来指定子进程的工作目录。如果没有指定，默认值是当前父进程的工作目录。

使用 `env` 来指定子进程中可用的环境变量，默认值是 `process.env` 。

Example of running ls -lh /usr, capturing stdout, stderr, and the exit code: 一个运行 `ls -lh /usr` ，获取 `stdout` ， `stderr` 和退出码得例子：

```
var spawn = require('child_process').spawn,
    ls    = spawn('ls', ['-lh', '/usr']);

ls.stdout.on('data', function (data) {
  console.log('stdout: ' + data);
});

ls.stderr.on('data', function (data) {
  console.log('stderr: ' + data);
});

ls.on('close', function (code) {
  console.log('child process exited with code ' + code);
});
```

例子：一个非常精巧的运行 `ps ax | grep ssh` 的方式

```
var spawn = require('child_process').spawn,
    ps    = spawn('ps', ['ax']),
    grep  = spawn('grep', ['ssh']);

ps.stdout.on('data', function (data) {
  grep.stdin.write(data);
});

ps.stderr.on('data', function (data) {
  console.log('ps stderr: ' + data);
});

ps.on('close', function (code) {
  if (code !== 0) {
    console.log('ps process exited with code ' + code);
  }
  grep.stdin.end();
});

grep.stdout.on('data', function (data) {
  console.log('' + data);
});

grep.stderr.on('data', function (data) {
  console.log('grep stderr: ' + data);
});

grep.on('close', function (code) {
  if (code !== 0) {
    console.log('grep process exited with code ' + code);
  }
});
```

一个检查执行失败的例子：

```
var spawn = require('child_process').spawn,
    child = spawn('bad_command');

child.on('error', function (err) {
  console.log('Failed to start child process.');
});
```

options.stdio

作为快捷方式， `stdio` 的值可以是一下字符串之一：

'pipe' - ['pipe', 'pipe', 'pipe'], 这是默认值 'ignore' - ['ignore', 'ignore', 'ignore'] 'inherit' - [process.stdin, process.stdout, process.stderr]或[0,1,2]

否则， `child_process.spawn()` 的 `stdio` 参数时一个数组，数组中的每一个索引的对应子进程中的一个文件标识符。可以是下列值之一：

'pipe' - 创建一个子进程与父进程之间的管道，管道的父进程端已父进程的 `child_process` 对象的属性（ `ChildProcess.stdio[fd]` ）暴露给父进程。为文件表示（fds）0 - 2 创建的管道也可以通过 `ChildProcess.stdin` ， `ChildProcess.stdout` 和 `ChildProcess.stderr` 分别访问。

'ipc' - 创建一个子进程和父进程间 传输信息/文件描述符 的IPC信道。一个子进程最多可能有一个IPC stdio 文件描述符。设置该选项将激活 `ChildProcess.send()` 方法。如果子进程向此文件描述符中写入JSON数据，则会触发 `ChildProcess.on('message')` 。如果子进程是一个 `node.js` 程序，那么IPC信道的存在将会激活 `process.send()` 和 `process.on('message')` 。

'ignore' - 不在子进程中设置文件描述符。注意 `node.js` 总是会为通过 `spawn` 创建的子进程打开文件描述符(fd) 0 - 2。如果这其中任意一项被设置为了 `ignore` ， `node.js` 会打开 `/dev/null` 并将其附给子进程对应的文件描述符（fd）。

Stream object - 与子进程共享一个与tty，文件，socket，或管道相关的可读/可写流。该流底层（underlying）的文件标识在子进程中被复制给stdio数组索引对应的文件描述符（fd）。

Positive integer - 该整形值被解释为父进程中打开的文件标识符。他与子进程共享，和Stream被共享的方式相似。

null, undefined - 使用默认值。For 对于stdio fds 0,1,2（或者说 `stdin`，`stdout` 和 `stderr` ），pipe管道被建立。对于fd 3及往后，默认为 `ignore` 。

例子：

```
var spawn = require('child_process').spawn;

// Child will use parent's stdios
spawn('prg', [], { stdio: 'inherit' });

// Spawn child sharing only stderr
spawn('prg', [], { stdio: ['pipe', 'pipe', process.stderr] });

// Open an extra fd=4, to interact with programs present a
// startd-style interface.
spawn('prg', [], { stdio: ['pipe', null, null, null, 'pipe'] });
```

options.detached

如果 `detached` 选项被设置，子进程将成为新进程组的领导。这使得在父进程退出后，子进程继续执行成为可能。

默认情况下，父进程会等待脱离了的子进程退出。要阻止父进程等待一个给出的子进程，请使用 `child.unref()` 方法，则父进程的事件循环的计数中将不包含这个子进程。

一个脱离的长时间运行的进程，以及将它的输出重定向到文件中的例子：

```
var fs = require('fs'),
    spawn = require('child_process').spawn,
    out = fs.openSync('./out.log', 'a'),
    err = fs.openSync('./out.log', 'a');

var child = spawn('prg', [], {
  detached: true,
  stdio: [ 'ignore', out, err ]
});

child.unref();
```

当使用 `detached` 选项创建一个长时间运行的进程时，进程不会保持运行除非向它提供了一个不连接到父进程的 `stdio` 的配置。如果继承了父进程的 `stdio` ，那么子进程将会继续附着在控制终端。

参阅： `child_process.exec()` 和 `child_process.fork()`

child_process.exec(command[, options], callback)

- command String 将要运行的命令，参数使用空格隔开
- **options Object**
  - cwd String 子进程的当前工作目录
  - env Object 环境变量键值对
  - encoding String 字符编码（默认： 'utf8'）
  - shell String 将要执行命令的Shell（默认: 在UNIX中为 `/bin/sh` ， 在Windows中为 `cmd.exe` ，Shell应当能识别 `-c` 开关在UNIX中，或 `/s /c` 在Windows中。 在Windows中，命令行解析应当能兼容 `cmd.exe` ）
  - timeout Number 超时时间（默认： 0）
  - maxBuffer Number 在stdout或stderr中允许存在的最大缓冲（二进制），如果超出那么子进程将会被杀死 （默认: 200*1024）
  - killSignal String 结束信号（默认：'SIGTERM'）
  - uid Number 设置用户进程的ID
  - gid Number 设置进程组的ID
- **callback Function**

    - error Error
    - stdout Buffer
    - stderr Buffer

- Return: ChildProcess object

在Shell中运行一个命令，并缓存命令的输出。

```
var exec = require('child_process').exec,
    child;

child = exec('cat *.js bad_file | wc -l',
  function (error, stdout, stderr) {
    console.log('stdout: ' + stdout);
    console.log('stderr: ' + stderr);
    if (error !== null) {
      console.log('exec error: ' + error);
    }
});
```

回调函数的参数是 `error` ， `stdout` ， `stderr` 。在成功时， `error` 将会是 `null` 。在发生错误时， `error` 将会是一个 `Error` 实例， `error.code` 将会是子进程的退出码， `error.signal` 将会被设置为结束进程的信号。

第二个可选的参数用于指定一些配置，默认值为：

```
{ encoding: 'utf8',
  timeout: 0,
  maxBuffer: 200*1024,
  killSignal: 'SIGTERM',
  cwd: null,
  env: null }
```

如果 `timeout` 大于0，那么子进程在运行时超过 `timeout` 时将会被杀死。子进程使用 `killSignal` 信号结束（默认为: 'SIGTERM'）。 `maxBuffer` 指定了 `stdout` ， `stderr` 中的最大数据量（字节），如果超过了这个数据量子进程也会被杀死。

注意：不像POSIX中的 `exec()` ， `child_process.exec()` 不替换已经存在的进程并且使用一个SHELL去执行命令。

child_process.execFile(file[, args][, options][, callback])

- file String 将要运行的命令，参数使用空格隔开
- args 字符串参数数组

- **options Object** -cwd String 子进程的当前工作目录 -env Object 环境变量键值对

  - encoding String 字符编码（默认： 'utf8'）
  - timeout Number 超时时间（默认： 0）
  - maxBuffer Number 在stdout或stderr中允许存在的最大缓冲（二进制），如果超出那么子进程将会被杀死 （默认: 200*1024）
  - killSignal String 结束信号（默认：'SIGTERM'）
  - uid Number 设置用户进程的ID
  - gid Number 设置进程组的ID
- **callback Function**

  - error Error
  - stdout Buffer
  - stderr Buffer
- Return: ChildProcess object

这个方法和 `child_process.exec()` 相似，除了它不是使用一个子SHELL执行命令而是直接执行文件。因此它比 `child_process.exec` 稍许精简一些。它们有相同的配置。

child_process.fork(modulePath[, args][, options])

- modulePath String 将要在子进程中运行的模块
- args Array 字符串参数数组

- **options Object**

- cwd String 子进程的当前工作目录
- env Object 环境变量键值对
- execPath String 创建子进程的可执行文件
- execArgv Array 子进程的可执行文件的字符串参数数组（默认：

process.execArgv）
- silent Boolean 如果为 `true` ，子进程的 `stdin` ， `stdout` 和 `stderr` 将会被关联至父进程，否则，它们将会从父进程中继承。（默认为： `false` ）
- uid Number 设置用户进程的ID
- gid Number 设置进程组的ID

Return: ChildProcess object

这个方法是 `spawn()` 的特殊形式，用于创建 `node.js` 进程。返回的对象除了拥有 `ChildProcess` 实例的所有方法，还有一个内建的通信信道。详情参阅 `child.send(message, [sendHandle])` 。

这些 `node.js` 子进程都是全新的V8实例。每个新的 `node.js` 进程都至少需要30ms启动以及10mb的内存。所以，你不能无休止地创建它们。

`options` 对象中的 `execPath` 属性可以用非当前 `node.js` 可执行文件来创建子进程。这需要小心使用，并且缺省情况下会使用子进程上的 `NODE_CHANNEL_FD` 环境变量所指定的文件描述符来通讯。该文件描述符的输入和输出假定为以行分割的JSON对象。

注意：不像POSIX中的 `fork()` ， `child_process.fork()` 不会复制当前进程。

同步进程创建

以下这些方法是同步的，意味着它们会阻塞事件循环。直到被创建的进程退出前，代码都将停止执行。

这些同步方法对简化大多数脚本任务都十分有用，并对简化应用配置的加载/执行也之分有用。

child_process.spawnSync(command[, args][, options])

- command 将要运行的命令
- args Array 字符串参数数组

- **options Object**
  - cwd String 子进程的当前工作目录
  - input String|Buffer 将要被作为 `stdin` 传入被创建的进程的值，提供这个值将会覆盖 `stdio[0]`

    - stdio Array 子进程的 `stdio` 配置
    - env Object 环境变量键值对
    - uid Number 设置用户进程的ID
    - gid Number 设置进程组的ID
    - timeout Number 毫秒数，子进程允许运行的最长时间（默认： `undefined` ）
    - killSignal String 结束信号（默认：'SIGTERM'）
    - maxBuffer Number 在stdout或stderr中允许存在的最大缓冲（二进制），如果超出那么子进程将会被杀死
    - encoding String 被用于所有 `stdio` 输入和输出的编码（默认：'buffer'）
- **return: Object**
    - pid Number 子进程的PID
    - output Array `stdio` 输出结果的数组
    - stdout Buffer|String 子进程 `stdout` 的内容
    - stderr Buffer|String 子进程 `stderr` 的内容
    - status Number 子进程的退出码
    - signal String 被用于杀死自进程的信号
    - error Error 若子进程运行失败或超时，它将会是对应的错误对象

`spawnSync` 会在子进程完全结束后才返回。当运行超时或被传递 `killSignal` 时，这个方法会等到进程完全退出才返回。也就是说，如果子进程处理了 `SIGTERM` 信号并且没有退出，你的父进程会继续阻塞。

child_process.execFileSync(command[, args][, options])

- command String 将要运行的命令
- args Array 字符串参数数组
- **options Object**
    - cwd String 子进程的当前工作目录
    - input String|Buffer 将要被作为 `stdin` 传入被创建的进程的值，提供这个值将会覆盖 `stdio[0]`
    - stdio Array 子进程的 `stdio` 配置（默认: 'pipe'）， `stderr` 默认得将会输出到父进程的 `stderr` ，除非指定了 `stdio`
    - env Object 环境变量键值对
    - uid Number 设置用户进程的ID

  - gid Number 设置进程组的ID
  - timeout Number 毫秒数，子进程允许运行的最长时间（默认： `undefined` ）
  - killSignal 结束信号（默认：'SIGTERM'）
  - maxBuffer Number 在stdout或stderr中允许存在的最大缓冲（二进制），如果超出那么子进程将会被杀死
  - encoding String 被用于所有 `stdio` 输入和输出的编码（默认：'buffer'）
- return: Buffer|String 此命令的 `stdout`

`execFileSync` 会在子进程完全结束后才返回。当运行超时或被传递 `killSignal` 时，这个方法会等到进程完全退出才返回。也就是说，如果子进程处理了 `SIGTERM` 信号并且没有退出，你的父进程会继续阻塞。

如果子进程超时或有一个非零的状态码，这个方法会抛出一个错误。这个错误对象与 `child_process.spawnSync` 的错误对象相同。

child_process.execSync(command[, options])

- command 将要运行的命令
- **options Object**
  - cwd String 子进程的当前工作目录
  - input String|Buffer 将要被作为 `stdin` 传入被创建的进程的值，提供这个值将会覆盖 `stdio[0]`
  - stdio Array 子进程的 `stdio` 配置（默认: 'pipe'）， `stderr` 默认得将会输出到父进程的 `stderr` ，除非指定了 `stdio`
  - env Object 环境变量键值对
  - uid Number 设置用户进程的ID
  - gid Number 设置进程组的ID
  - timeout Number 毫秒数，子进程允许运行的最长时间（默认： `undefined` ）
  - killSignal String 结束信号（默认：'SIGTERM'）
  - maxBuffer Number 在stdout或stderr中允许存在的最大缓冲（二进制），如果超出那么子进程将会被杀死
  - encoding String 被用于所有 `stdio` 输入和输出的编码（默认：'buffer'）
- return: Buffer|String 此命令的 `stdout`

`execSync` 会在子进程完全结束后才返回。当运行超时或被传递 `killSignal` 时，这个方法会等到进程完全退出才返回。也就是说，如果子进程处理了 `SIGTERM` 信号并且没有退出，你的父进程会继续阻塞。

如果子进程超时或有一个非零的状态码，这个方法会抛出一个错误。这个错误对象与 `child_process.spawnSync` 的错误对象相同。

Cluster

稳定度: 2 - 稳定

单个的 `node.js` 实例运行在单线程上。为了享受多核系统的优势，用户需要启动一个 `node.js` 集群来处理负载。

`cluster` 模块允许你方便地创建共享服务器端口的子进程：

```
var cluster = require('cluster');
var http = require('http');
var numCPUs = require('os').cpus().length;

if (cluster.isMaster) {
  // Fork workers.
  for (var i = 0; i < numCPUs; i++) {
    cluster.fork();
  }

  cluster.on('exit', function(worker, code, signal) {
    console.log('worker ' + worker.process.pid + ' died');
  });
} else {
  // Workers can share any TCP connection
  // In this case its a HTTP server
  http.createServer(function(req, res) {
    res.writeHead(200);
    res.end("hello world\n");
  }).listen(8000);
}
```

启动 `node.js` 将会在工作线程中共享8000端口：

```
% NODE_DEBUG=cluster iojs server.js
23521,Master Worker 23524 online
23521,Master Worker 23526 online
23521,Master Worker 23523 online
23521,Master Worker 23528 online
```

这个特性是最近才开发的，并且可能在未来有所改变。请试用它并提供反馈。

注意，在Windows中，在工作进程中建立命名管道服务器目前是不可行的。

工作原理

工作进程通过 `child_process.fork` 方法被创建，所以它们可以与父进程通过IPC管道沟通以及相互传递服务器句柄。

集群模式支持两种分配传入连接的方式。

第一种（并且是除了Windows平台外默认的方式）是循环式。主进程监听一个端口，接受新连接，并且以轮流的方式分配给工作进程，并且以一些内建机制来避免一个工作进程过载。

第二种方式是，主进程建立监听 `socket` 并且将它发送给感兴趣的工作进程。工作进程直接接受传入的连接。

第二种方式理论上有最好的性能。但是在实践中，操作系统的调度不可预测，分配往往十分不平衡。负载曾被观察到8个进程中，超过70%的连接结束于其中的2个进程。

因为 `server.listen()` 将大部分工作交给了主进程，所以一个普通的 `node.js` 进程和一个集群工作进程会在三种情况下有所区别：

1. `server.listen({fd: 7})` 因为消息被传递给了主进程，主进程的文件描述符 `7` 会被监听，并且句柄会被传递给工作进程而不是监听工作进程中文件描述符为 `7` 的东西。

2. `server.listen(handle)` 明确地监听句柄，会让工作进程使用给定的句柄，而不是与主进程通信。如果工作进程已经有了此句柄，那么将假设你知道你在做什么。

3. `server.listen(0)` 通常，这会导致服务器监听一个随机端口。但是，在集群中，每次调用 `listen(0)` 时，每一个工作进程会收到同样的“随机”端口。也就是说，端口只是在第一次方法被调用时是随机的，但在之后是可预知的。如果你想监听特定的端口，则根据工作进程的PID来生成端口号。

由于在 `node.js` 或你的程序中的工作进程间没有路由逻辑也没有共享的状态。所以，请不要为你的程序设计成依赖太重的内存数据对象，如设计会话和登陆时。

因为工作进程都是独立的进程，它们可以根据你程序的需要被杀死或被创建，并且并不会影响到其他工作进程。只要有活跃的工作进程，那么服务器就会继续接收连接。但是 `node.js` 不会自动地为你管理工作进程数。所以根据你的应用需求来管理工作进程池是你的责任。

cluster.schedulingPolicy

调度策略，选择 `cluster.SCHED_RR` 来使用循环式，或选择 `cluster.SCHED_NONE` 来由操作系统处理。这是一个全局设定，并且在你第一次启动了一个工作进程或调用 `cluster.setupMaster()` 方法后就不可再更改。

`SCHED_RR` 是除了Windows外其他操作系统中的默认值。一旦 `libuv` 能够有效地分配IOCP句柄并且没有巨大的性能损失，那么Windows下的默认值也会变为它。

`cluster.schedulingPolicy` 也可以通过环境变量 `NODE_CLUSTER_SCHED_POLICY` 来设定。合法值为 `rr` 和 `none` 。

cluster.settings

- **Object**
  - execArgv Array 传递给 `node.js` 执行的字符串参数（默认为 `process.execArgv` ）
  - exec String 工作进程文件的路径（默认为 `process.argv[1]` ）
  - args Array 传递给工作进程的字符串参数（默认为 `process.argv.slice(2)` ）
  - silent Boolean 是否将工作进程的输出传递给父进程的 `stdio` （默认为 `false` ）
  - uid Number 设置用户进程的ID
  - gid Number 设置进程组的ID

在调用 `.setupMaster()` （或 `.fork()` ）方法之后，这个 `settings` 对象会存放方法的配置，包括默认值。

因为 `.setupMaster()` 仅能被调用一次，所以这个对象被设置后便不可更改。

这个对象不应由你来手工更改或设置。

cluster.isMaster

- Boolean

如果进程是主进程则返回 `true` 。这由 `process.env.NODE_UNIQUE_ID` 决定。如果 `process.env.NODE_UNIQUE_ID` 为 `undefined` ，那么就返回 `true` 。

cluster.isWorker

- Boolean

如果进程不是主进程则返回 `true` 。

Event: 'fork'

- worker Worker object

当一个新的工作进程由 `cluster` 模块所开启时会触发 `fork` 事件。这能被用来记录工作进程活动日志，或创建自定义的超时。

```
var timeouts = [];
function errorMsg() {
  console.error("Something must be wrong with the connection ..."
);
}

cluster.on('fork', function(worker) {
  timeouts[worker.id] = setTimeout(errorMsg, 2000);
});
cluster.on('listening', function(worker, address) {
  clearTimeout(timeouts[worker.id]);
});
cluster.on('exit', function(worker, code, signal) {
  clearTimeout(timeouts[worker.id]);
  errorMsg();
});
```

Event: 'online'

- worker Worker object

当创建了一个新的工作线程后，工作线程必须响应一个在线信息。当主进程接收到在线信息后它会触发这个事件。`fork` 和 `online` 事件的区别在于：`fork` 是主进程创建了工作进程后触发，`online` 是工作进程开始运行时触发。

```
cluster.on('online', function(worker) {
  console.log("Yay, the worker responded after it was forked");
});
```

Event: 'listening'

- worker Worker object
- address Object

当工作进程调用 `listen()` 方法。服务器会触发 `listening` 事件，集群中的主进程也会触发一个 `listening` 事件。

这个事件的回调函数包含两个参数，第一个 `worker` 是一个包含工作进程的对象， `address` 对象是一个包含以下属性的对象： `address` ， `port` 和 `addressType` 。当工作进程监听多个地址时，这非常有用。

```
cluster.on('listening', function(worker, address) {
  console.log("A worker is now connected to " + address.address
+ ":" + address.port);
});
```

`addressType` 是以下中的一个：

- 4 (TCPv4)
- 6 (TCPv6)
- -1 (unix domain socket)
- "udp4" 或 "udp6" (UDP v4 或 v6)

Event: 'disconnect'

- worker Worker object

当工作进程的IPC信道断开连接时触发。这个事件当工作进程优雅地退出，被杀死，或手工断开连接（如调用 `worker.disconnect()` ）后触发。

`disconnect` 和 `exit` 事件之间可能存在延迟。这两个事件可以用来侦测是否进程在清理的过程中被阻塞，或者是否存在长连接。

```
cluster.on('disconnect', function(worker) {
  console.log('The worker #' + worker.id + ' has disconnected');
});
```

Event: 'exit'

- worker Worker object
- code Number 如果正常退出，则为退出码
- signal String 导致进程被杀死的信号的信号名（如'SIGHUP'）

当任何一个工作进程结束时， `cluster` 模块会触发一个 `exit` 事件。

这可以被用来通过再次调用 `.fork()` 方法重启服务器。

```
cluster.on('exit', function(worker, code, signal) {
  console.log('worker %d died (%s). restarting...',
    worker.process.pid, signal || code);
  cluster.fork();
});
```

参阅 `child_process` 事件： `exit` 。

Event: 'setup'

- settings Object

每次 `.setupMaster()` 方法被调用时触发。

这个 `settings` 对象与 `.setupMaster()` 被调用时 `cluster.settings` 对象相同，并且仅供查询，因为 `.setupMaster()` 可能在一次事件循环里被调用多次。

如果保持精确十分重要，请使用 `cluster.settings` 。

cluster.setupMaster([settings])

- **settings Object**
  - exec String 工作进程文件的路径（默认为 `process.argv[1]` ）
  - args Array 传递给工作进程的参数字符串（默认为 `process.argv.slice(2)` ）
  - silent Boolean 是否将工作进程的输出传递给父进程的 `stdio` (默认为 `false`)

`setupMaster` 方法被用来改变默认的 `fork` 行为。一旦被调用， `settings` 参数将被表现为 `cluster.settings` 。

注意：

- 任何 `settings` 的改变仅影响之后的 `.fork()` 调用，而不影响已经运行中的工作进程
- 工作进程中唯一不同通过 `.setupMaster()` 来设置的属性是传递给 `.fork()` 方法的 `env` 参数
- 上文中的参数的默认值仅在第一次调用时被应用，之后的调用的默认值是当

前 `cluster.setupMaster()` 被调用时的值。

例子：

```
var cluster = require('cluster');
cluster.setupMaster({
  exec: 'worker.js',
  args: ['--use', 'https'],
  silent: true
});
cluster.fork(); // https worker
cluster.setupMaster({
  args: ['--use', 'http']
});
cluster.fork(); // http worker
```

这只能被主进程调用。

cluster.fork([env])

- env Object 将添加到工作进程环境变量的键值对
- return Worker object

创建一个新的工作进程。

这只能被主进程调用。

cluster.disconnect([callback])

- callback Function 当所有的工作进程断开连接并且所有句柄关闭后调用

在 `cluster.workers` 中的每一个工作进程中调用 `.disconnect()` 。

当所有进程断开连接，所有内部的句柄都将关闭，如果没有其他的事件处于等待，将允许主进程优雅地退出。

这个方法接受一个可选的将会在结束时触发的回调函数参数。

这只能被主进程调用。

cluster.worker

- Object

当前工作进程对象的引用。对于主进程不可用。

```
var cluster = require('cluster');

if (cluster.isMaster) {
  console.log('I am master');
  cluster.fork();
  cluster.fork();
} else if (cluster.isWorker) {
  console.log('I am worker #' + cluster.worker.id);
}
```

cluster.workers

- Object

一个储存了所有活跃的工作进程对象的哈希表，以 `id` 字段为主键。这使得遍历所有工作进程变得容易。仅在主进程中可用。

当工作进程断开连接或退出时，它会从 `cluster.workers` 中移除。这个两个事件的触发顺序不能被提前决定。但是，能保证的是，从 `cluster.workers` 移除一定发生在这两个事件触发之后。

```
// Go through all workers
function eachWorker(callback) {
  for (var id in cluster.workers) {
    callback(cluster.workers[id]);
  }
}
eachWorker(function(worker) {
  worker.send('big announcement to all workers');
});
```

想要跨越通信信道来得到一个工作进程的引用时，使用工作进程的唯一 `id` 能简单找到工作进程。

```
socket.on('data', function(id) {
  var worker = cluster.workers[id];
});
```

Class: Worker

`Worker` 对象包含了一个工作进程所有的公开信息和方法。在主进程中它可以通过 `cluster.workers` 取得。在工作进程中它可以通过 `cluster.worker` 取得。

worker.id

- String

每一个新的工作进程都被给予一个独一无二的id，这个id被存储在此 `id` 属性中。

当一个工作进程活跃时，这是它被索引在 `cluster.workers` 中的主键。

worker.process

- ChildProcess object

所有的工作进程都通过 `child_process.fork()` 被创建，返回的对象被作为 `.process` 属性存储。在一个工作进程中，全局的 `process` 被存储。

参阅 `Child Process module`

注意，如果在进程中 `disconnect` 事件触发并且 `.suicide` 属性不为 `true` ，那么进程会调用 `process.exit(0)` 。这防止了意外的断开连接。

worker.suicide

- Boolean

通过调用 `.kill()` 或 `.disconnect()` 设置，在这之前他为 `undefined` 。

布尔值 `worker.suicide` 使你可以区别自发和意外的退出，主进程可以通过这个值来决定使用重新创建一个工作进程。

```
cluster.on('exit', function(worker, code, signal) {
  if (worker.suicide === true) {
    console.log('Oh, it was just suicide\' – no need to worry').
  }
});

// kill worker
worker.kill();
```

worker.send(message[, sendHandle])

- message Object
- sendHandle Handle object

给工作进程或主进程发生一个信息，可选得添加一个句柄。

在主进程中它将给特定的工作进程发送一个信息。它指向 `child.send()` 。

在工作进程中它将给主进程发送一个信息。它指向 `process.send()` 。

下面的例子将来自主进程的所有信息返回：

```
if (cluster.isMaster) {
  var worker = cluster.fork();
  worker.send('hi there');

} else if (cluster.isWorker) {
  process.on('message', function(msg) {
    process.send(msg);
  });
}
```

worker.kill([signal='SIGTERM'])

- signal String 传递给工作进程的结束信号名

这个函数将会杀死工作进程。在主进程中，它通过断开 `worker.process` 做到，并且一旦断开，使用 `signal` 杀死进程。在工作进程中，它通过断开信道做到，然后使用退出码 `0` 退出。

会导致 `.suicide` 被设置。

为了向后兼任，这个方法的别名是 `worker.destroy()` 。

注意在工作进程中， `process.kill()` 存在，但它不是这个函数。是 `process.kill(pid[, signal])` 。

worker.disconnect()

在工作进程中，这个函数会关闭所有的服务器，等待这些服务器上的 `close` 事件，然后断开IPC信道。

在主进程中，一个内部信息会被传递给工作进程，至使它们自行调用 `.disconnect()` 。

会导致 `.suicide` 被设置。

注意在一个服务器被关闭后，它将不会再接受新连接，但是连接可能被其他正在监听的工作进程所接收。已存在的连接将会被允许向往常一样退出。当没有更多的连接存在时，工作进程的IPC信道会关闭并使之优雅地退出，参阅 `server.close()` 。

以上说明仅应用于服务器连接，客户端连接将不会自动由工作进程关闭，并且在退出前，不会等到连接退出。

注意在工作进程中， `process.disconnect` 存在，但它不是这个函数。是 `child.disconnect()` 。

由于长连接可能会阻塞工作进程的退出，这时传递一个动作信息非常有用，应用来根据信息指定的动作来关闭它们。超时机制是上述的有用实现，在 `disconnect` 事件在指定时长后没有触发时，杀死工作进程。

```
if (cluster.isMaster) {
  var worker = cluster.fork();
  var timeout;

  worker.on('listening', function(address) {
    worker.send('shutdown');
    worker.disconnect();
    timeout = setTimeout(function() {
      worker.kill();
    }, 2000);
  });

  worker.on('disconnect', function() {
    clearTimeout(timeout);
  });

} else if (cluster.isWorker) {
  var net = require('net');
  var server = net.createServer(function(socket) {
    // connections never end
  });

  server.listen(8000);

  process.on('message', function(msg) {
    if(msg === 'shutdown') {
      // initiate graceful close of any connections to server
    }
  });
}
```

worker.isDead()

如果工作进程已经被关闭，则返回 `true` 。

worker.isConnected()

如果工作进程通过它的IPC信道连接到主进程，则返回 `true` 。一个工作进程在被创建后连接到它的主进程。在 `disconnect` 事件触发后它会断开连接。

Event: 'message'

- message Object

这个事件与 `child_process.fork()` 所提供的事件完全相同。

在工作进程中你也可以使用 `process.on('message')` 。

例子，这里有一个集群，使用消息系统在主进程中统计请求的数量：

```
var cluster = require('cluster');
var http = require('http');

if (cluster.isMaster) {

  // Keep track of http requests
  var numReqs = 0;
  setInterval(function() {
    console.log("numReqs =", numReqs);
  }, 1000);

  // Count requestes
  function messageHandler(msg) {
    if (msg.cmd && msg.cmd == 'notifyRequest') {
      numReqs += 1;
    }
  }

  // Start workers and listen for messages containing notifyRequ
est
  var numCPUs = require('os').cpus().length;
  for (var i = 0; i < numCPUs; i++) {
    cluster.fork();
  }

  Object.keys(cluster.workers).forEach(function(id) {
    cluster.workers[id].on('message', messageHandler);
  });

} else {
```

```
  // Worker processes have a http server.
  http.Server(function(req, res) {
    res.writeHead(200);
    res.end("hello world\n");

    // notify master about the request
    process.send({ cmd: 'notifyRequest' });
  }).listen(8000);
}
```

Event: 'online'

与 `cluster.on('online')` 事件相似，但指向了特定的工作进程。

```
cluster.fork().on('online', function() {
  // Worker is online
});
```

这不是在工作进程中触发的。

Event: 'listening'

- address Object

与 `cluster.on('listening')` 事件相似，但指向了特定的工作进程。

```
cluster.fork().on('listening', function(address) {
  // Worker is listening
});
```

这不是在工作进程中触发的。

Event: 'disconnect'

与 `cluster.on('disconnect')` 事件相似，但指向了特定的工作进程。

```
cluster.fork().on('disconnect', function() {
  // Worker has disconnected
});
```

Event: 'exit'

- code Number 如果正常退出，则为退出码
- signal String 导致进程被杀死的信号名（如 `'SIGHUP'` ）

与 `cluster.on('exit')` 事件相似，但指向了特定的工作进程。

```
var worker = cluster.fork();
worker.on('exit', function(code, signal) {
  if( signal ) {
    console.log("worker was killed by signal: "+signal);
  } else if( code !== 0 ) {
    console.log("worker exited with error code: "+code);
  } else {
    console.log("worker success!");
  }
});
```

Event: 'error'

这个事件与 `child_process.fork()` 所提供的事件完全相同。

在工作进程中你也可以使用 `process.on('error')` 。

Console

稳定度: 2 - 稳定

这个模块定义了一个控制台类，并且暴露了一个 `console` 对象。

`console` 对象是一个特殊的 `Console` 实例，它的输出被传至 `stdout` 或 `stderr` 。

为了使用的方便， `console` 被定义为一个全局对象，不需要通过 `require` 就可直接使用。

console

- Object

用来向 `stdout` 和 `stderr` 打印信息。与大多数浏览器提供的 `console` 对象的功能类似，只是这里输出被传至 `stdout` 或 `stderr` 。

当目的地是终端或文件时（为了避免过早退出丢失信息）， `console` 函数时同步的。当目的地是管道时（为了避免长时间阻塞）， `console` 函数时异步的。

下面的例子里， `stdout` 是非阻塞的， `stderr` 是阻塞的：

```
$ node script.js 2> error.log | tee info.log
```

日常使用时，除了你需要记录大量数量的数据，你不用担心阻塞/非阻塞。

console.log([data][, ...])

向 `stdout` 打印一行新信息。这个函数可以像 `printf()` 那样接受多个参数，例子：

```
var count = 5;
console.log('count: %d', count);
// prints 'count: 5'
```

如果第一个字符串中没有发现格式化元素，那么 `util.inspect` 将被应用到各个参数。详情参阅 `util.format()` 。

console.info([data][, ...])

与 `console.log` 相同。

console.error([data][, ...])

与 `console.log` 相同。但是输出至 `stderr` 。

console.warn([data][, ...])

与 `console.err` 相同。

console.dir(obj[, options])

对 `obj` 调用 `util.inspect` 并且将结果字符串输出至 `stdout` 。这个函数会忽略 `obj` 上的任何自定义 `inspect()` 函数。一个可选的 `options` 参数可以被传递用来格式化字符串的某些方面：

- showHidden - 如果为 `true` ， `object` 的不可枚举和标志属性也会被显示。默认为 `false` 。
- depth - 告诉 `inspect` 在格式化对象时递归多少次。在检查大而复杂的对象时很有用。默认为2。若要递归到底则传递 `null` 。
- colors - 如果为 `true` ，那么输出会以ANSI颜色码的形式输出。默认为 `false` 。颜色是可以自定义，参阅下文。

console.time(label)

被用来计算指定操作之间时间间隔。为了开始一个 `timer` ，调用 `console.time()` 方法，作为唯一参数可以给它一个名字。为了关闭一个 `timer` ，并且得到毫秒间隔，仅仅以相同的名字参数调用一次 `console.timeEnd()` 。

console.timeEnd(label)

停止一个之前通过 `console.time()` 开启的 `timer` ，并且向控制台打印结果。

例子：

```
console.time('100-elements');
for (var i = 0; i < 100; i++) {
  ;
}
console.timeEnd('100-elements');
// prints 100-elements: 262ms
```

console.trace(message[, ...])

向 `stderr` 打印 `'Trace :'` ，跟随着格式化信息和堆栈信息。

console.assert(value[, message][, ...])

与 `assert.ok()` 类似，但是错误信息被像 `util.format(message...)` 一样格式化。

Class: Console

使用 `require('console')` 后。 `Console` 或 `console.Console` 可以取得这个类。

```
var Console = require('console').Console;
var Console = console.Console;
```

你可以调用 `Console` 类来自定义如 `console` 一样的简单日记记录器，但是有不同的输出流。

new Console(stdout[, stderr])

通过传递一个或两个可写流实例，创建一个新的 `Console` 。 `stdout` 是一个用来打印日志和信息的输出流。 `stderr` 是一个被用来打印警告和错误输出的。如果 `stderr` 没有被传递，那么警告和错误信息将被传递至 `stdout` 。

```
var output = fs.createWriteStream('./stdout.log');
var errorOutput = fs.createWriteStream('./stderr.log');
// custom simple logger
var logger = new Console(output, errorOutput);
// use it like console
var count = 5;
logger.log('count: %d', count);
// in stdout.log: count 5
```

全局的 `console` 是一个特殊的 `Console` 实例，它的输出被传递至 `process.stdout` 和 `process.stderr` ：

```
new Console(process.stdout, process.stderr);
```

Crypto

稳定度: 2 - 稳定

使用 `require('crypto')` 来获取这个模块。

`crypto` 模块提供了一种封装安全证书的方法，用来作为安全HTTPS网络和HTTP链接的一部分。

它也提供了一个OpenSSL `hash` ，`hamc` ，`cipher` ，`decipher` ，`sign` 和 `vierify` 方法的包装集合。

crypto.setEngine(engine[, flags])

加载和设置 一些/所有 OpenSSL功能引擎（由标记选择）。

引擎可以通过id或 引擎共享库的路径 来选择。

`flags` 是可选的，并且有一个 `ENGINE_METHOD_ALL` 默认值。可以选一个或多个以下的标记（在常量模块中定义）。

- ENGINE_METHOD_RSA
- ENGINE_METHOD_DSA
- ENGINE_METHOD_DH
- ENGINE_METHOD_RAND
- ENGINE_METHOD_ECDH
- ENGINE_METHOD_ECDSA
- ENGINE_METHOD_CIPHERS
- ENGINE_METHOD_DIGESTS
- ENGINE_METHOD_STORE
- ENGINE_METHOD_PKEY_METH
- ENGINE_METHOD_PKEY_ASN1_METH
- ENGINE_METHOD_ALL
- ENGINE_METHOD_NONE

crypto.getCiphers()

返回一个支持的加密算法的名字数组。

例子：

```
var ciphers = crypto.getCiphers();
console.log(ciphers); // ['aes-128-cbc', 'aes-128-ccm', ...]
```

crypto.getHashes()

返回一个支持的哈希算法的名字数组。

例子：

```
var hashes = crypto.getHashes();
console.log(hashes); // ['sha', 'sha1', 'sha1WithRSAEncryption',
 ...]
```

crypto.getCurves()

返回一个支持的椭圆加密算法的名字数组。

例子：

```
var curves = crypto.getCurves();
console.log(curves); // ['secp256k1', 'secp384r1', ...]
```

crypto.createCredentials(details)

> 稳定度: 0 - 弃用。使用 `tls.createSecureContext` 代替。

创建一个加密凭证对象，接受一个可选的带键字典 `details` ：

- pfx : 一个带着 `PFX` 或 `PKCS12` 加密的私钥，加密凭证和CA证书的字符串或 `buffer` 。
- key : 一个带着 `PEM` 加密私钥的字符串。
- passphrase : 一个私钥或 `pfx` 密码字符串。
- cert : 一个带着 `PEM` 加密凭证的字符串。
- ca : 一个用来信任的 `PEM` 加密CA证书的字符串或字符串列表。

- crl : 一个 `PEM` 加密 `CRL` 的字符串或字符串列表。
- ciphers: 一个描述需要使用或排除的加密算法的字符串。更多加密算法的格式细节参阅 `http://www.openssl.org/docs/apps/ciphers.html#CIPHER_LIST_FORMAT`

如果没有指定 `ca` ，那么 `node.js` 将会使用 `http://mxr.mozilla.org/mozilla/source/security/nss/lib/ckfw/builtins/certdata.txt.` 提供的默认公共可信任 `CA` 列表。

crypto.createHash(algorithm)

创建并返回一个哈希对象，一个指定算法的加密哈希用来生成哈希摘要。

`algorithm` 依赖于平台上的OpenSSL版本所支持的算法。例如 `'sha1'` ， `'md5'` ， `'sha256'` ， `'sha512'` 等等。 `openssl list-message-digest-algorithms` 命令会展示可用的摘要算法。

例子：这个程序计算出一个文件的sha1摘要：

```
var filename = process.argv[2];
var crypto = require('crypto');
var fs = require('fs');

var shasum = crypto.createHash('sha1');

var s = fs.ReadStream(filename);
s.on('data', function(d) {
  shasum.update(d);
});

s.on('end', function() {
  var d = shasum.digest('hex');
  console.log(d + '  ' + filename);
});
```

Class: Hash

这个类用来创建数据哈希摘要。

这是一个同时可读与可写的流。写入的数据用来计算哈希。一旦当流的可写端终止，使用 `read()` 来获取计算所得哈希摘要。遗留的 `update` 和 `digest` 方法同样被支持。

通过 `crypto.createHash` 返回。

hash.update(data[, input_encoding])

使用给定的 `data` 更新哈希内容，通过 `input_encoding` 指定的编码可以是 `'utf8'` ， `'ascii'` 或 `'binary'` 。如果没有提供编码，并且输入是一个字符串，那么将会指定编码为 `'binary'` 。如果 `data` 是一个 `Buffer` 那么 `input_encoding` 会被忽略。

它是流式数据，所以这个方法可以被调用多次。

hash.digest([encoding])

计算所有的被传递的数据的摘要。 `encoding` 可以是 `'binary'` ， `'hex'` 或 `'base64'` 。如果没有指定编码，那么一个 `buffer` 被返回。

注意：当调用了 `digest()` 方法之后，哈希对象不能再被使用了。

crypto.createHmac(algorithm, key)

创建并返回一个hmac对象，即通过给定的算法和密钥生成的加密图谱（cryptographic）。

这是一个既可读又可写的流。写入的数据被用来计算hamc。一旦当流的可写端终止，使用 `read()` 方法来获取计算所得摘要值。遗留的 `update` 和 `digest` 方法同样被支持。

`algorithm` 依赖于平台上的OpenSSL版本所支持的算法。参阅上文 `createHash` 。 `key` 是要使用的hmac密钥。

Class: Hmac

用于创建hmac加密图谱（cryptographic）的类。

通过 `crypto.createHmac` 返回。

hmac.update(data)

只用指定的 `data` 更新hmac内容。因为它是流式数据，所以这个方法可以被调用多次。

hmac.digest([encoding])

计算所有的被传递的数据的hmac摘要。 `encoding` 可以是 `'binary'` ， `'hex'` 或 `'base64'` 。如果没有指定编码，那么一个 `buffer` 被返回。

注意：当调用了 `digest()` 方法之后，hmac对象不能再被使用了。

crypto.createCipher(algorithm, password)

创建和返回一个 `cipher` 对象，指定指定的算法和密码。

算法依赖于OpenSSL，如果 `'aes192'` ，等等。在最近的发行版中， `openssl list-cipher-algorithms` 命令会展示可用的 `cipher` 算法。密码被用来获取密钥和IV，必须是一个 `'binary'` 编码的字符串或 `buffer` 。

这是一个既可读又可写的流。写入的数据被用来计算哈希。一旦当流的可写端终止，使用 `read()` 方法来获取通过 `cipher` 计算所得的内容。遗留的 `update` 和 `digest` 方法同样被支持。

注意： `createCipher` 通过 无盐 `MD5` 一次迭代所得的摘要 来调用 `OpenSSL函数` EVP_BytesToKey` 来派生密钥。无盐意味允许字典攻击，即同样的密码经常可以用来创建同样的密钥。一次迭代并且无加密图谱安全（non-cryptographically secure）以为着允许密码被快速测试。

OpenSSL建议使用 `pbkdf2` 替代 `EVP_BytesToKey` ，推荐你通过 `crypto.pbkdf2` 然后调用 `createCipheriv()` 创建一个 `cipher` 流来派生一个密钥和iv。

crypto.createCipheriv(algorithm, key, iv)

创建和返回一个 `cipher` 对象，指定指定的算法，密钥和iv。

`algorithm` 参数与 `createCipher()` 相同。 `key` 是被算法使用的源密钥（raw key）。iv是初始化矢量（initialization vector）。

`key` 和 `iv` 必须是 `'binary'` 编码的字符串或 `buffer` 。

Class: Cipher

创建一个加密数据。

由 `crypto.createCipher` 和 `crypto.createCipheriv` 返回。

这是一个既可读又可写的流。写入的文本数据被用来在可读端生产被加密的数据。遗留的 `update` 和 `final` 方法同样被支持。

cipher.update(data[, input_encoding][, output_encoding])

通过 `data` 更新 `cipher` ， `input_encoding` 中指定的编码可以是 `'utf8'` ， `'ascii'` 或 `'binary'` 。如果没有提供编码，那么希望接受到一个 `buffer` 。如果数据是一个 `Buffer` ，那么 `input_encoding` 将被忽略。

`output_encoding` 指定了加密数据的输出格式，可以是 `'binary'` ， `'base64'` 或 `'hex'` 。如果没有指定编码，那么一个 `buffer` 会被返回。

返回一个加密内容，并且因为它是流式数据，所以可以被调用多次。

cipher.final([output_encoding])

返回所有的剩余的加密内容， `output_encoding` 可以是 `'binary'` ， `'base64'` 或 `'hex'` 。如果没有指定编码，那么一个 `buffer` 会被返回。

注意：当调用了 `final()` 方法之后，cipher对象不能再被使用了。

cipher.setAutoPadding(auto_padding=true)

你可以禁用自动填充输入数据至块大小。如果 `auto_padding` 为 `false` ，那么整个输入数据的长度必须 `cipher` 的块大小的整数倍，否则会失败。这对非标准填充非常有用，如使用0x0替代PKCS填充。你必须在 `cipher.final` 之前调用它。

cipher.getAuthTag()

对于已认证加密模式（当前支持：GCM），这个方法返回一个从给定数据计算所得的代表了认证标签的 `Buffer` 。必须在 `final` 方法被调用后调用。

cipher.setAAD(buffer)

对于已认证加密模式（当前支持：GCM），这个方法设置被用于额外已认证数据（AAD）输入参数的值。

crypto.createDecipher(algorithm, password)

使用给定算法和密钥，创建并返回一个解密器对象。这是上文 `createCipher()` 的一个镜像。

crypto.createDecipheriv(algorithm, key, iv)

使用给定算法，密钥和iv，创建并返回一个解密器对象。这是上文 `createCipheriv()` 的一个镜像。

Class: Decipher

解密数据类。

通过 `crypto.createDecipher` 和 `crypto.createDecipheriv` 返回。

这是一个既可读又可写的流。写入的被加密的数据被用来在可读端生产文本数据。遗留的 `update` 和 `final` 方法同样被支持。

decipher.update(data[, input_encoding][, output_encoding])

通过 `data` 更新 `decipher` ，编码可以是 `'binary'` ， `'base64'` 或 `'hex'` 。如果没有提供编码，那么希望接受到一个 `buffer` 。如果数据是一个 `Buffer` ，那么 `input_encoding` 将被忽略。

`output_encoding` 指定了解密数据的输出格式，可以是 `'binary'` ， `'ascii'` 或 `'utf8'` 。如果没有指定编码，那么一个 `buffer` 会被返回。

decipher.final([output_encoding])

返回所有的剩余的文本数据，`output_encoding` 可以是 `'binary'` ，`'ascii'` 或 `'utf8'` 。如果没有指定编码，那么一个 `buffer` 会被返回。

注意：当调用了 `final()` 方法之后，decipher对象不能再被使用了。

decipher.setAutoPadding(auto_padding=true)

如数据没有使用标准块填充阻止 `decipher.final` 检查和删除它来加密，你可以禁用自动填充。那么整个输入数据的长度必须 `cipher` 的块大小的整数倍，否则会失败。你必须在将数据导流至 `decipher.update` 前调用它。

decipher.setAuthTag(buffer)

对于已认证加密模式（当前支持：GCM），这个方法必须被传递，用来接受认证标签。如果没有提供标签或密文被干扰，最终会抛出一个错误。

decipher.setAAD(buffer)

对于已认证加密模式（当前支持：GCM），这个方法设置被用于额外已认证数据（AAD）输入参数的值。

crypto.createSign(algorithm)

使用指定的算法，创建并返回一个数字签名类。在最近的OpenSSL发行版中， `openssl list-public-key-algorithms` 会列出所有支持的数字签名算法。例如 `'RSA-SHA256'` 。

Class: Sign

用于生成数字签名的类。

通过 `crypto.createSign` 返回。

`Sign` 对象是一个可写流。写入的数据用来生成数字签名。一旦所有的数据被写入， `sign` 方法会返回一个数字签名。遗留的 `update` 方法也支持。

sign.update(data)

使用 `data` 更新 `sign` 对象。因为它是流式的所以这个方法可以被调用多次。

sign.sign(private_key[, output_format])

根据所有通过 `update` 方法传入的数据计算数字签名。

`private_key` 可以是一个对象或一个字符串，如果 `private_key` 是一个字符串，那么它被当做没有密码的密钥。

private_key:

- key : 包含 `PEM` 编码私钥的字符串。
- passphrase : 一个私钥密码的字符串。

返回的数字签名编码由 `output_format` 决定，可以是 `'binary'` ，`'hex'` 或 `'base64'` 。如果没有指定编码，会返回一个 `buffer` 。

注意，在调用了 `sign()` 后，`sign` 对象不能再使用了。

crypto.createVerify(algorithm)

使用给定的算法，创建并返回一个验证器对象。这个对象是 `sign` 对象的镜像。

Class: Verify

用来验证数字签名的类。

由 `crypto.createVerify` 返回。

`Verify` 对象是一个可写流。写入的数据用来验证提供的数字签名。一旦所有的数据被写入，`verify` 方法会返回 `true` 如果提供的数字签名有效。遗留的 `update` 方法也支持。

verifier.update(data)

使用 `data` 更新 `verifier` 对象。因为它是流式的所以这个方法可以被调用多次。

verifier.verify(object, signature[, signature_format])

通过使用 `object` 和 `signature` 验证被签名的数据。 `object` 是一个包含了PEM编码对象的字符串，这个对象可以是RSA公钥，DSA公钥或X.509证书。 `signature` 是先前计算出来的数字签名， `signature_format` 可以是 `'binary'` ， `'hex'` 或 `'base64'` 。如果没有指定编码，那么希望收到一个 `buffer` 。

返回值是 `true` 或 `false` 根据数字签名对于数据和公钥的有效性。

注意，在调用了 `verify()` 后， `verifier` 对象不能再使用了。

crypto.createDiffieHellman(prime_length[, generator])

创建一个迪菲－赫尔曼密钥交换对象（Diffie-Hellman key exchange object），并且根据 `prime_length` 生成一个质数，可以指定一个可选的数字生成器。如果没有指定生成器，将使用 `2` 。

crypto.createDiffieHellman(prime[, prime_encoding][, generator][, generator_encoding])

通过给定的质数，和可选的生成器，创建一个迪菲－赫尔曼密钥交换对象（Diffie-Hellman key exchange object）。 `generator` 可以是一个数字，字符串或 `Buffer` 。如果没有指定生成器，将使用 `2` 。 `prime_encoding` 和 `generator_encoding` 可以是 `'binary'` ， `'hex'` 或 `'base64'` 。如果没有指定 `prime_encoding` ，那么希望 `prime` 是一个 `Buffer` 。如果没有指定 `generator_encoding` ，那么希望 `generator` 是一个 `Buffer` 。

Class: DiffieHellman

叫来创建迪菲－赫尔曼密钥交换的类。

通过 `crypto.createDiffieHellman` 返回。

diffieHellman.verifyError

一个包含了所有警告和/或错误的位域，作为检查初始化时的执行结果。以下是这个属性的合法属性（被常量模块定义）：

- DH_CHECK_P_NOT_SAFE_PRIME

- DH_CHECK_P_NOT_PRIME
- DH_UNABLE_TO_CHECK_GENERATOR
- DH_NOT_SUITABLE_GENERATOR

diffieHellman.generateKeys([encoding])

生成一个 私和公 迪菲－赫尔曼 密钥值，并且返回一个指定编码的公钥。这个密钥可以被转移给第三方。编码可以是 `'binary'` ， `'hex'` 或 `'base64'` 。如果没有提供编码，那么会返回一个 `buffer` 。

diffieHellman.computeSecret(other_public_key[, input_encoding][, output_encoding])

使用 `other_public_key` 作为第三方密钥来计算共享秘密（shared secret），并且返回计算结果。提供的密钥会以 `input_encoding` 来解读，并且秘密以 `output_encoding` 来编码。编码可以是 `'binary'` ， `'hex'` 或 `'base64'` 。如果没有提供编码，那么会返回一个 `buffer` 。

如果没有指定 `output_encoding` ，那么会返回一个 `buffer` 。

diffieHellman.getPrime([encoding])

根据指定编码返回一个迪菲－赫尔曼质数，编码可以是 `'binary'` ， `'hex'` 或 `'base64'` 。如果没有提供编码，那么会返回一个 `buffer` 。

diffieHellman.getGenerator([encoding])

根据指定编码返回一个迪菲－赫尔曼生成器，编码可以是 `'binary'` ， `'hex'` 或 `'base64'` 。如果没有提供编码，那么会返回一个 `buffer` 。

diffieHellman.getPublicKey([encoding])

根据指定编码返回一个迪菲－赫尔曼公钥，编码可以是 `'binary'` ， `'hex'` 或 `'base64'` 。如果没有提供编码，那么会返回一个 `buffer` 。

diffieHellman.getPrivateKey([encoding])

根据指定编码返回一个迪菲－赫尔曼私钥，编码可以是 `'binary'` ，`'hex'` 或 `'base64'` 。如果没有提供编码，那么会返回一个 `buffer` 。

diffieHellman.setPublicKey(public_key[, encoding])

设置迪菲－赫尔曼公钥，密钥编码可以是 `'binary'` ，`'hex'` 或 `'base64'` 。如果没有提供编码，那么期望接收一个 `buffer` 。

diffieHellman.setPrivateKey(private_key[, encoding])

设置迪菲－赫尔曼私钥，密钥编码可以是 `'binary'` ，`'hex'` 或 `'base64'` 。如果没有提供编码，那么期望接收一个 `buffer` 。

crypto.getDiffieHellman(group_name)

创建一个预定义的迪菲－赫尔曼密钥交换对象。支持的群组有：'modp1', 'modp2', 'modp5' (由RFC 2412定义) 和 'modp14', 'modp15', 'modp16', 'modp17', 'modp18' (由RFC 3526定义)。返回的对象模仿 `crypto.createDiffieHellman()` 创建的对象的借口，但是不允许交换密钥（如通过 `diffieHellman.setPublicKey()` ）。执行这套流程的好处是双方不需要事先生成或交换组余数，节省了处理和通信时间。

例子（获取一个共享秘密）：

```
var crypto = require('crypto');
var alice = crypto.getDiffieHellman('modp5');
var bob = crypto.getDiffieHellman('modp5');

alice.generateKeys();
bob.generateKeys();

var alice_secret = alice.computeSecret(bob.getPublicKey(), null,
'hex');
var bob_secret = bob.computeSecret(alice.getPublicKey(), null, '
hex');

/* alice_secret and bob_secret should be the same */
console.log(alice_secret == bob_secret);
```

crypto.createECDH(curve_name)

使用由 `curve_name` 指定的预定义椭圆，创建一个椭圆曲线（EC）迪菲—赫尔曼密钥交换对象。使用 `getCurves()` 来获取可用的椭圆名列表。在最近的发行版中， `openssl ecparam -list_curves` 命令也会展示可用的椭圆曲线的名字和简述。

Class: ECDH

用于EC迪菲—赫尔曼密钥交换的类。

由 `crypto.createECDH` 返回。

ECDH.generateKeys([encoding[, format]])

生成一个 私/公 EC迪菲—赫尔曼密钥值，并且返回指定格式和编码的公钥。这个密钥可以被转移给第三方。

`format` 指定点的编码，可以是 `'compressed'` ， `'uncompressed'` 或 `'hybrid'` 。如果没有指定，那么点将是 `'uncompressed'` 格式。

编码可以是 `'binary'` ， `'hex'` 或 `'base64'` 。如果没有提供编码，那么会返回一个 `buffer` 。

ECDH.computeSecret(other_public_key[, input_encoding][, output_encoding])

使用 `other_public_key` 作为第三方密钥来计算共享秘密（shared secret），并且返回计算结果。提供的密钥会以 `input_encoding` 来解读，并且秘密以 `output_encoding` 来编码。编码可以是 `'binary'` ， `'hex'` 或 `'base64'` 。如果没有提供编码，那么会返回一个 `buffer` 。

如果没有指定 `output_encoding` ，那么会返回一个 `buffer` 。

ECDH.getPublicKey([encoding[, format]])

返回指定编码和格式的EC迪菲－赫尔曼公钥。

`format` 指定点的编码，可以是 `'compressed'` ， `'uncompressed'` 或 `'hybrid'` 。如果没有指定，那么点将是 `'uncompressed'` 格式。

编码可以是 `'binary'` ， `'hex'` 或 `'base64'` 。如果没有提供编码，那么会返回一个 `buffer` 。

ECDH.getPrivateKey([encoding])

返回指定编码的EC迪菲－赫尔曼私钥，编码可以是 `'binary'` ， `'hex'` 或 `'base64'` 。如果没有提供编码，那么会返回一个 `buffer` 。

ECDH.setPublicKey(public_key[, encoding])

设置EC迪菲－赫尔曼公钥。密钥编码可以是 `'binary'` ， `'hex'` 或 `'base64'` 。如果没有提供编码，那么期望接收一个 `buffer` 。

ECDH.setPrivateKey(private_key[, encoding])

设置EC迪菲—赫尔曼私钥。密钥编码可以是 `'binary'` ， `'hex'` 或 `'base64'` 。如果没有提供编码，那么期望接收一个 `buffer` 。

例子（获取一个共享秘密）：

```
var crypto = require('crypto');
var alice = crypto.createECDH('secp256k1');
var bob = crypto.createECDH('secp256k1');

alice.generateKeys();
bob.generateKeys();

var alice_secret = alice.computeSecret(bob.getPublicKey(), null,
'hex');
var bob_secret = bob.computeSecret(alice.getPublicKey(), null, '
hex');

/* alice_secret and bob_secret should be the same */
console.log(alice_secret == bob_secret);
```

crypto.pbkdf2(password, salt, iterations, keylen[, digest], callback)

异步PBKDF2函数。提供被选择的HAMC摘要函数（默认为SHA1）来获取一个请求长度的密码密钥，盐和迭代数。回调函数有两个参数：（ `err` ， `derivedKey` ）。

例子：

```
crypto.pbkdf2('secret', 'salt', 4096, 512, 'sha256', function(er
r, key) {
  if (err)
    throw err;
  console.log(key.toString('hex'));  // 'c5e478d...1469e50'
});
```

可用通过 `crypto.getHashes()` 获取支持的摘要函数列表。

crypto.pbkdf2Sync(password, salt, iterations, keylen[, digest])

同步PBKDF2函数。返回 `derivedKey` 或抛出错误。

crypto.randomBytes(size[, callback])

生成有密码图谱一般健壮的伪随机数据，用处：

```
// async
crypto.randomBytes(256, function(ex, buf) {
  if (ex) throw ex;
  console.log('Have %d bytes of random data: %s', buf.length, bu
f);
});

// sync
try {
  var buf = crypto.randomBytes(256);
  console.log('Have %d bytes of random data: %s', buf.length, bu
f);
} catch (ex) {
  // handle error
  // most likely, entropy sources are drained
}
```

注意：如果熵不足，那么它会阻塞。尽管它从不话费超过几毫秒。唯一可以想到的阻塞是情况是，当整个系统的熵还是很低时，在其之后启动。

Class: Certificate

这个类用来处理已签名公钥 & 挑战（challenges）。最常用的是它的一系列处理 `<keygen>` 元素的函数。 `http://www.openssl.org/docs/apps/spkac.html` 。

通过 `crypto.Certificate` 返回。

Certificate.verifySpkac(spkac)

返回 `ture` 或 `false` ，依赖于SPKAC的有效性。

Certificate.exportChallenge(spkac)

导出编码好的公钥从指定的SPKAC。

Certificate.exportPublicKey(spkac)

导出编码好的挑战（challenge）从指定的SPKAC。

crypto.publicEncrypt(public_key, buffer)

使用 `public_key` 加密 `buffer` 。目前只支持RSA。

`public_key` 可是是一个对象或一个字符串。如果 `public_key` 是一个字符串，它会被视作没有密码的密钥并且将使用 `RSA_PKCS1_OAEP_PADDING` 。因为 `RSA` 公钥可以用来从你传递给这个方法的密钥来获取。

public_key:

- key : 一个包含PEM加密的私钥字符串
- passphrase : 一个可选的私钥密码字符串
- **padding** : 一个可选的填充值，以下值之一：
    - constants.RSA_NO_PADDING
    - constants.RSA_PKCS1_PADDING
    - constants.RSA_PKCS1_OAEP_PADDING

注意：所有的填充值都被常量模块所定义。

crypto.publicDecrypt(public_key, buffer)

详情参阅上文。与 `crypto.publicEncrypt` 有相同API。默认填充值是 `RSA_PKCS1_PADDING` 。

crypto.privateDecrypt(private_key, buffer)

使用 `private_key` 解密 `buffer` 。

`private_key` 可以是一个对象或一个字符串。如果 `private_key` 是一个字符串，它会当做没有密码的密钥，并且使用 `RSA_PKCS1_OAEP_PADDING` 。

public_key:

- key : 一个包含PEM加密的私钥字符串
- passphrase : 一个可选的私钥密码字符串
- **padding** : 一个可选的填充值，以下值之一：
  - constants.RSA_NO_PADDING
  - constants.RSA_PKCS1_PADDING
  - constants.RSA_PKCS1_OAEP_PADDING

注意：所有的填充值都被常量模块所定义。

crypto.privateEncrypt(private_key, buffer)

详情参阅上文。与 `crypto.privateDecrypt` 有相同API。默认填充值是 `RSA_PKCS1_PADDING` 。

crypto.DEFAULT_ENCODING

默认编码是用于接受字符串或 `buffer` 的函数。默认值是 `'buffer'` ，所以默认是使用 `Buffer` 对象的。这被用来与旧的以 `'binary'` 为默认编码的程序更好地兼容。

注意新的程序仍可能期望使用 `buffer` ，所以只将它作为一个临时措施。

近期的API改变

`Crypto` 模块在还没有统一的流API概念，以及没有 `Buffer` 对象来处理二进制数据前就加入了 `Node.js` 。

因为这样，它的流类没有其他 `node.js` 类的典型类，而且很多方法默认接受和返回二进制字符串而不是 `Buffer` 。这些函数将被改成默认接受和返回 `Buffer` 。

这对于一些但不是所有的使用场景来说是巨大的改变。

例如，如果你现在对 `Sign` 类使用默认参数，并且传递 `Verify` 类的结果，不检查数据，那么在以前它将会继续工作。在你曾经得到二进制字符串的地方，你将会得到一个 `Buffer` 。

但是，如果你正在使用那些使用字符串可以，但使用 `Buffer` 不能工作的数据（如连接它们，存储进数据库等）。或者对 `crypto` 函数不传递编码参数来传递二进制字符串。那么以后，你需要提供你想要指定的编码。如果要将默认的使用风格，转换为旧风格的话，将 `crypto.DEFAULT_ENCODING` 域设置为 `'binary'` 。注意新的程序仍可能期望接受 `buffer` ，所以这仅作为一个临时措施。

Debugger

稳定度: 2 - 稳定

V8自带了一个强大的调试器，可以从外部通过TCP协议访问。 `node.js` 为这个调试器内建了一个客户端。要使用它的话，使用 `debug` 参数启动 `node.js` ；会出现提示符：

```
% iojs debug myscript.js
< debugger listening on port 5858
connecting... ok
break in /home/indutny/Code/git/indutny/myscript.js:1
  1 x = 5;
  2 setTimeout(function () {
  3   debugger;
debug>
```

`node.js` 的调试器客户端并未支持所有的命令，但是简单的步进和调试都是可以的。通过在源代码就放置 `debugger;` 语句，你可以启用一个断点。

例如，假设又一个这样的 `myscript.js` ：

```
// myscript.js
x = 5;
setTimeout(function () {
  debugger;
  console.log("world");
}, 1000);
console.log("hello");
```

那么一旦你打开调试器，它会在第四行中断。

```
% iojs debug myscript.js
< debugger listening on port 5858
connecting... ok
break in /home/indutny/Code/git/indutny/myscript.js:1
  1 x = 5;
  2 setTimeout(function () {
  3   debugger;
debug> cont
< hello
break in /home/indutny/Code/git/indutny/myscript.js:3
  1 x = 5;
  2 setTimeout(function () {
  3   debugger;
  4   console.log("world");
  5 }, 1000);
debug> next
break in /home/indutny/Code/git/indutny/myscript.js:4
  2 setTimeout(function () {
  3   debugger;
  4   console.log("world");
  5 }, 1000);
  6 console.log("hello");
debug> repl
Press Ctrl + C to leave debug repl
> x
5
> 2+2
4
debug> next
< world
break in /home/indutny/Code/git/indutny/myscript.js:5
  3   debugger;
  4   console.log("world");
  5 }, 1000);
  6 console.log("hello");
  7
debug> quit
%
```

`repl` 命令允许你远程地执行代码。 `next` 命令步进到下一行。还有一些其他的可用命令，输入 `help` 查看它们。

Watchers

在调试代码时，你可监视表达式和变量的值。在每个断点，监视器列表上的每个表达式会被在当前上下文执行，并且断点的源代码前展示。

为了开始监视一个表达式，输入 `watch("my_expression")` 。 `watchers` 打印可用的监视器。为了移除一个监视器，输入 `unwatch("my_expression")` 。

命令参考

Stepping

- cont, c - 继续执行
- next, n - 下一步
- step, s - 介入（Step in）
- out, o - 离开（Step out）
- pause - 暂停代码执行（类似开发者工具中的暂停按钮）

Breakpoints

- setBreakpoint(), sb() - 在当前行设置一个断点
- setBreakpoint(line), sb(line) - 在指定行设置一个断点
- setBreakpoint('fn()'), sb(...) - 在函数体的第一个语句上设置断点
- setBreakpoint('script.js', 1), sb(...) - 在 `script.js` 的第一行设置断点
- clearBreakpoint('script.js', 1), cb(...) - 清除 `script.js` 第一行的断点

同样也可以在一个还未载入的文件（模块）中设置断点：

```
% ./iojs debug test/fixtures/break-in-module/main.js
< debugger listening on port 5858
connecting to port 5858... ok
break in test/fixtures/break-in-module/main.js:1
  1 var mod = require('./mod.js');
  2 mod.hello();
  3 mod.hello();
debug> setBreakpoint('mod.js', 23)
Warning: script 'mod.js' was not loaded yet.
  1 var mod = require('./mod.js');
  2 mod.hello();
  3 mod.hello();
debug> c
break in test/fixtures/break-in-module/mod.js:23
 21
 22 exports.hello = function() {
 23   return 'hello from module';
 24 };
 25
debug>
```

Info

- **backtrace, bt** - 打印当前执行框架的回溯
- **list(5)** - 列出脚本源代码的5行上下文（前5行和后5行）
- **watch(expr)** - 为监视列表添加表达式
- **unwatch(expr)** - 从监视列表中移除表达式
- **watchers** - 列出所有的监视器和它们的值（会在每一个断点自动列出）
- **repl** - 在所调试的脚本上下文中打开调试器的 `repl`

Execution control

- **run** - 运行脚本（在调试器开始时自动运行）
- **restart** - 重启脚本
- **kill** - 结束脚本

Various

- **scripts** - 列出所有载入的脚本

- version - 展示V8版本

高级使用

V8调试器可以通过 使用 `--debug` 命令行参数打开 `node.js` 或 向一个已存在的 `node.js` 进程发送 `SIGUSR1` 信号 来启用。

一旦一个进程被设置为了调试模式，它就可以被连接到 `node.js` 调试器。可以通过pid或URI来连接，语法为：

- iojs debug -p - 通过pid连接进程
- iojs debug - 通过URI（如 `localhost:5858` ）连接进程

DNS

稳定度: 2 - 稳定

通过 `require('dns')` 来获取这个模块。

这个模块包含以下两类函数：

1) 使用底层操作系统工具来进行域名解析的函数，并且不需要进行任何网络活动。这类函数只有一个： `dns.lookup` 。希望与 在其他操作系统的其他应用 执行域名解析 有相同行为时，请使用 `dns.lookup` 。

下面是一个解析 `www.google.com` 的例子：

```
var dns = require('dns');

dns.lookup('www.google.com', function onLookup(err, addresses, family) {
  console.log('addresses:', addresses);
});
```

2) 连接实际的DNS服务器来进行域名解析的函数，并且经常使用网络来执行DNS查找。除了 `dns.lookup` 外 `DNS` 模块的所有函数都属于这类。这类函数不与 `dns.lookup` 使用相同的配置文件。例如，它们不使用 `/etc/hosts` 配置文件。这类函数适合那些不希望使用底层操作系统工具来进行域名解析，总是想要执行DNS查询的开发者。

下面例子是，解析 `'www.google.com'` ，然后反向解析返回的IP地址。

```
var dns = require('dns');

dns.resolve4('www.google.com', function (err, addresses) {
  if (err) throw err;

  console.log('addresses: ' + JSON.stringify(addresses));

  addresses.forEach(function (a) {
    dns.reverse(a, function (err, hostnames) {
      if (err) {
        throw err;
      }

      console.log('reverse for ' + a + ': ' + JSON.stringify(hos
tnames));
    });
  });
});
```

两者之间的选择会产生微妙的结果，更多信息请查询下文 `实现注意事项` 章节。

dns.lookup(hostname[, options], callback)

解析 `hostname` （如 `'google.com'` ）为第一个找到的A（IPv4）或AAAA（IPv6）记录。 `options` 可以是对象或者数组。如果 `options` 没有提供，那么IPv4和IPv6都是有效的。如果 `options` 是一个数组，那么它必须是 `4` 或 `6` 。

另外， `options` 可以是一个含有以下属性的对象：

- family: {Number} - 地址族。如果提供，必须为整数 `4` 或 `6` 。如果没有提供，那么IPv4和IPv6都是有效的。
- hints: {Number} - 如果提供，它必须是一个或多个支持的 `getaddrinfo` 标识。如果没有提供，那么没有标识被传递给 `getaddrinfo` 。多个标识可以通过在逻辑上 `ORing` 它们的值，来传递给 `hints` 。支持的 `getaddrinfo` 标识请参阅下文。
- all: {Boolean} - 如果 `true` ，那么回调函数以数组的形式返回所有解析的地址，否则只返回一个地址。默认为 `false` 。

所有的属性都是可选的，以下是一个 `options` 例子：

```
{
  family: 4,
  hints: dns.ADDRCONFIG | dns.V4MAPPED,
  all: false
}
```

回调函数有参数（err, address, family）。 `address` 是IPv4或IPv6地址字符串。 `family` 是 `adress` 的协议族，即 `4` 或 `6` 。

如果 `options` 的所有参数都被设置，那么参数转变为（err, addresses），`addresses` 是一个地址和协议族数组。

若发生错误， `err` 是错误对象， `err.code` 是错误码。不仅在 `hostname` 不存在时，在如没有可用的文件描述符等情况下查找失败， `err.code` 也会被设置为 `'ENOENT'` 。

`dns.lookup` 不需要与DNS协议有任何关系。它仅仅是一个连接名字和地址的操作系统功能。

在任何的 `node.js` 程序中，它的实现对表现有一些微妙但是重要的影响。在使用前，请花一些时间查阅 `实现注意事项` 章节。

dns.lookupService(address, port, callback)

解析给定的 `address` 和 `port` 为一个主机名和使用 `getnameinfo` 的服务。

回调函数有参数（err, hostname, service）。 `hostname` 和 `service` 参数是字符串（如分别为 `'localhost'` 和 `'http'` ）。

若发生错误， `err` 是错误对象， `err.code` 是错误码。

dns.resolve(hostname[, rrtype], callback)

使用指定的 `rrtype` 类型，解析主机名（如 `'google.com'` ）为一个记录数组。

有效的 `rrtype` 有：

- 'A' (IPV4 地址，默认)
- 'AAAA' (IPV6 地址)

- 'MX' (邮件交换记录)
- 'TXT' (文本记录)
- 'SRV' (SRV记录)
- 'PTR' (用于IP反向查找)
- 'NS' (域名服务器记录)
- 'CNAME' (别名记录)
- 'SOA' (权限开始记录)

回调函数有参数（err, addresses）。`address` 中每个元素的类型由记录类型所指定，并且在下文相应的查找方法中有描述。

若发生错误，`err` 是错误对象，`err.code` 是下文错误代码列表中的一个。

dns.resolve4(hostname, callback)

与 `dns.resolve()` 相同，但只使用IPv4查询（一个记录）。地址是一个IPv4地址数组（如 `['74.125.79.104', '74.125.79.105', '74.125.79.106']`）。

dns.resolve6(hostname, callback)

与 `dns.resolve4()` 相同，除了使用IPv6查询（一个AAAA查询）。

dns.resolveMx(hostname, callback)

与 `dns.resolve()` 相同，但是只用于邮件交换查询（MX记录）。

地址是一个MX记录数组，每一个元素都有一个 `priority` 和一个 `exchange` 属性（如 `[{'priority': 10, 'exchange': 'mx.example.com'},...]`）。

dns.resolveTxt(hostname, callback)

与 `dns.resolve()` 相同，但是只用于文本查询（TXT记录）。地址是一个 `hostname` 可用的2-d数组（如 `[ ['v=spf1 ip4:0.0.0.0 ', '~all' ] ]`）。每个字数组包含一个记录的TXT数据块。根据使用场景的不同，它们可能被连接在一起也可能被分开。

dns.resolveSrv(hostname, callback)

与 `dns.resolve()` 相同，但是只用于服务查询（SRV记录）。地址是一个 `hostname` 可用的SRV记录数组。SRV记录的属性有 `priority` ， `weight` ， `port` ，和 `name` （如 `[{'priority': 10, 'weight': 5, 'port': 21223, 'name': 'service.example.com'}, ...]` ）。

dns.resolveSoa(hostname, callback)

与 `dns.resolve()` 相同，但是只用于权限记录查询（SOA记录）。

地址是一个有以下结构的对象：

```
{
  nsname: 'ns.example.com',
  hostmaster: 'root.example.com',
  serial: 2013101809,
  refresh: 10000,
  retry: 2400,
  expire: 604800,
  minttl: 3600
}
```

dns.resolveNs(hostname, callback)

与 `dns.resolve()` 相同，但是只用于域名服务器查询（NS记录）。地址是一个 `hostname` 可用的域名服务器记录数组（如 `['ns1.example.com', 'ns2.example.com']` ）。

dns.resolveCname(hostname, callback)

与 `dns.resolve()` 相同，但是只用于别名记录（别名记录）。地址是一个 `hostname` 可用的别名数组（如 `['bar.example.com']` ）。

dns.reverse(ip, callback)

为得到一个主机名数组，反向查询一个IP。

回调函数有参数（err, hostnames）。

若发生错误，`err` 是错误对象，`err.code` 是下文错误代码列表中的一个。

dns.getServers()

返回一个正在被用于解析的IP地址字符串数组。

dns.setServers(servers)

给定一个IP地址字符串数组，将它们设置给用来解析的服务器。

如果你为地址指定了一个端口，端口会被忽略，因为底层库不支持。

如果你传递了非法输入，会抛出错误。

Error codes

每一次DNS查询都可能返回以下错误码之一：

- dns.NODATA: DNS服务器返回一个没有数据的应答。
- dns.FORMERR: DNS服务器声明查询是格式错误的。
- dns.SERVFAIL: DNS服务器返回一个普通错误。
- dns.NOTFOUND: 域名没有找到。
- dns.NOTIMP: DNS服务器没有实现请求的操作。
- dns.REFUSED: DNS服务器拒绝查询。
- dns.BADQUERY: 格式错误的DNS查询。
- dns.BADNAME: 格式错误的主机名。
- dns.BADFAMILY: 不支持的协议族。
- dns.BADRESP: 格式错误的DNS响应。
- dns.CONNREFUSED: 不能连接到DNS服务器。
- dns.TIMEOUT: 连接DNS服务器超时。
- dns.EOF: 文件末端。
- dns.FILE: 读取文件错误。
- dns.NOMEM: 内存溢出。
- dns.DESTRUCTION: 通道被销毁。
- dns.BADSTR: 格式错误的字符串。
- dns.BADFLAGS: 指定了非法标志。
- dns.NONAME: 给定的主机名不是数字。
- dns.BADHINTS: 给定的提示标识非法。

- dns.NOTINITIALIZED: `c-ares` 库初始化未被执行。
- dns.LOADIPHLPAPI: 加载 `iphlpapi.dll` 错误。
- dns.ADDRGETNETWORKPARAMS: 找不到 `GetNetworkParams` 函数。
- dns.CANCELLED: DNS查询被取消。

支持的getaddrinfo标识

以下标识可以被传递给 `dns.lookup` 的 `hints` ：

- dns.ADDRCONFIG: 返回的地址类型由当前系统支持的地址类型决定。例如，如果当前系统至少有一个IPv4地址被配置，那么将只会返回IPv4地址。回溯地址不被考虑。
- dns.V4MAPPED: 如果IPv6协议族被指定，但是没有发现IPv6地址，那么返回IPv6地址的IPv4映射。

实现注意事项

尽管 `dns.lookup` 和 `dns.resolve*/dns.reverse` 函数都用于关联一个域名和一个地址（或反之亦然），它们的行为还是有些许区别。这些差别虽然微小，但是对于 `node.js` 程序的行为有重大影响。

dns.lookup

在引擎下， `dns.lookup` 使用了和其他程序相同的操作系统功能。例如， `dns.lookup` 将总是和 `ping` 命令一样解析一个给定的域名。在大多类POSIX操作系统上， `dns.lookup` 函数的表现可以通过改变 `nsswitch.conf(5)` 和/或 `resolv.conf(5)` 的设置来调整，但是需要小心的是，改变这些文件将会影响这个操作系统上正在运行的所有其他程序。

虽然在 `JavaScript` 的角度，这个调用是异步的，但是它在libuv线程池中的实现是同步调用 `getaddrinfo(3)` 。因为libuv线程池有一个固定的大小，意味着如果 `getaddrinfo(3)` 花费了太多的时间，那么其他libuv线程池中的操作（如文件系统操作）会感觉到性能下降。为了缓解这个情况，一个潜在的解决方案是通过设置 `'UV_THREADPOOL_SIZE'` 环境变量大于4（当前默认值）来增加libuv线程池的大小。更多libuv线程池的信息，请参阅官方的libuv文档。

dns.resolve，以dns.resolve和dns.reverse开头的函数

这些函数的实现与 `dns.lookup` 相当不同。它们不使用 `getaddrinfo(3)` 并且它们总是通过网络执行一次DNS查询。这些网络通信通常是异步的，并且不使用libuv线程池。

作为结果，其他使用libuv线程池的 `dns.lookup` 方法的进程可能会有相同的负面影响，但这些函数没有。

它们与 `dns.lookup` 使用了不同的配置文件。例如，它们不使用 `/etc/hosts` 中的配置。

Errors

`node.js` 生成的错误分为两类： `JavaScript` 错误和系统错误。所有的错误都继承于 `JavaScript` 的 `Error` 类，或就是它的实例。并且都至少提供这个类中可用的属性。

当一个操作因为语法错误或语言运行时级别（language-runtime-level）的原因不被允许时，一个 `JavaScript error` 会被生成并抛出一个异常。如果一个操作因为系统级别（system-level）限制而不被允许时，一个系统错误会被生成。客户端代码接着会根据API传播它的方式来被给予捕获这个错误的机会。

API被调用的风格决定了生成的错误如何回送（handed back），传播给客户端。这反过来告诉客户端如何捕获它们。异常可以通过 `try / catch` 结构捕获；其他的捕获方式请参阅下文。

JavaScript错误

`JavaScript` 错误表示API被错误的使用了，或者正在写的程序有问题。

Class: Error

一个普通的错误对象。和其他的错误对象不同， `Error` 实例不指示任何 为什么错误发生 的原因。 `Error` 在它们被实例化时，会记录下“堆栈追踪”信息，并且可以会提供一个错误描述。

注意： `node.js` 会将系统错误以及 `JavaScript` 错误都封装为这个类的实例。

new Error(message)

实例化一个新的 `Error` 对象，并且用提供的 `message` 设置它的 `.message` 属性。它的 `.stack` 属性将会描述 `new Error` 被调用时程序的这一刻。堆栈追踪信息隶属于V8堆栈追踪API。堆栈追踪信息只延伸到同步代码执行的开始，或 `Error.stackTraceLimit` 给出的帧数（number of frames），这取决于哪个更小。

error.message

一个在 `Error()` 实例化时被传递的字符串。这个信息会出现在堆栈追踪信息的第一行。改变这个值将不会改变堆栈追踪信息的第一行。

error.stack

这个属性返回一个代表错误被实例化时程序运行的那个点的字符串。

一个堆栈追踪信息例子：

```
Error: Things keep happening!
   at /home/gbusey/file.js:525:2
   at Frobnicator.refrobulate (/home/gbusey/business-logic.js:42
4:21)
   at Actor.<anonymous> (/home/gbusey/actors.js:400:8)
   at increaseSynergy (/home/gbusey/actors.js:701:6)
```

第一行被格式化为 `<错误类名>: <错误信息>` ，然后是一系列的堆栈信息帧（以 `“at”` 开头）。每帧都描述了一个最终导致错误生成的一次调用的地点。V8会试图去给出每个函数的名字（通过变量名，函数名或对象方法名），但是也有可能它找不到一个合适的名字。如果V8不能为函数定义一个名字，那么那一帧里只会展示出位置信息。否则，被定义的函数名会显示在位置信息之前。

帧只会由 `JavaScript` 函数生成。例如，如果在一个 `JavaScript` 函数里，同步执行了一个叫 `cheetahify` 的C++ `addon` 函数，那么堆栈追踪信息中的帧里将不会有 `cheetahify` 调用：

```
var cheetahify = require('./native-binding.node');

function makeFaster() {
  // cheetahify *synchronously* calls speedy.
  cheetahify(function speedy() {
    throw new Error('oh no!');
  });
}

makeFaster(); // will throw:
// /home/gbusey/file.js:6
//     throw new Error('oh no!');
//           ^
// Error: oh no!
//     at speedy (/home/gbusey/file.js:6:11)
//     at makeFaster (/home/gbusey/file.js:5:3)
//     at Object.<anonymous> (/home/gbusey/file.js:10:1)
//     at Module._compile (module.js:456:26)
//     at Object.Module._extensions..js (module.js:474:10)
//     at Module.load (module.js:356:32)
//     at Function.Module._load (module.js:312:12)
//     at Function.Module.runMain (module.js:497:10)
//     at startup (node.js:119:16)
//     at node.js:906:3
```

位置信息将会是以下之一：

- `native` ，如果帧代表了向V8内部的一次调用（如在 `[].forEach` 中）。
- `plain-filename.js:line:column` ，如果帧代表了向 `node.js` 内部的一次调用。
- `/absolute/path/to/file.js:line:column` ，如果帧代表了向用户程序或其依赖的一次调用。

关键的一点是，代表了堆栈信息的字符串只在需要被使用时生成，它是惰性生成的。

堆栈信息的帧数由 `Error.stackTraceLimit` 或 当前事件循环的 `tick` 里可用的帧数 中小的一方决定。

系统级别错误被作为增强的 `Error` 实例生成，参阅下文。

Error.captureStackTrace(targetObject[, constructorOpt])

为 `targetObject` 创建一个 `.stack` 属性，它代表了 `Error.captureStackTrace` 被调用时，在程序中的位置。

```
var myObject = {};

Error.captureStackTrace(myObject);

myObject.stack  // similar to `new Error().stack`
```

追踪信息的第一行，将是 `targetObject.toString()` 的结果，而不是一个带有 `ErrorType:` 前缀的信息。

可选的 `constructorOpt` 接收一个函数。如果指定，所有 `constructorOpt` 以上的帧，包括 `constructorOpt` ，将会被生成的堆栈追踪信息忽略。

这对于向最终用户隐藏实现细节十分有用。一个普遍的使用这个参数的例子：

```
function MyError() {
  Error.captureStackTrace(this, MyError);
}

// without passing MyError to captureStackTrace, the MyError
// frame would should up in the .stack property. by passing
// the constructor, we omit that frame and all frames above it.

new MyError().stack
```

Error.stackTraceLimit

一个决定了堆栈追踪信息的堆栈帧数的属性（不论是由 `new Error().stack` 或由 `Error.captureStackTrace(obj)` 生成）。

初始值是 `10` 。可以被设置为任何有效的 `JavaScript` 数字，当值被改变后，就会影响所有的堆栈追踪信息的获取。如果设置为一个非数字值，堆栈追踪将不会获取任何一帧，并且会在要使用时报告 `undefined` 。

Class: RangeError

一个 `Error` 子类，表明了为一个函数提供的参数没有在可接受的值的范围之内；不论是在一个数字范围之外，或是在一个参数指定的参数集合范围之外。例子：

```
require('net').connect(-1);  // throws RangeError, port should be > 0 && < 65536
```

`node.js` 会立刻生成并抛出一个 `RangeError` 实例 -- 它们是参数验证的一种形式。

Class: TypeError

一个 `Error` 子类，表明了提供的参数不是被允许的类型。例如，为一个期望收到字符串参数的函数，传入一个函数作为参数，将导致一个类型错误。

```
require('url').parse(function() { }); // throws TypeError, since it expected a string
```

`node.js` 会立刻生成并抛出一个 `TypeError` 实例 -- 它们是参数验证的一种形式。

Class: ReferenceError

一个 `Error` 子类，表明了试图去获取一个未定义的对象的属性。大多数情况下它表明了一个输入错误，或者一个不完整的程序。客户端代码可能会生成和传播这些错误，但实际上只有V8会。

```
doesNotExist; // throws ReferenceError, doesNotExist is not a variable in this program.
```

`ReferenceError` 实例将有一个 `.arguments` 属性，它是一个包含了一个元素的数组。这个元素表示没有被定义的那个变量。

```
try {
  doesNotExist;
} catch(err) {
  err.arguments[0] === 'doesNotExist';
}
```

除非用户程序是动态生成并执行的，否则， `ReferenceErrors` 应该永远被认为是程序或其依赖模块的bug。

Class: SyntaxError

一个 `Error` 子类，表明了程序代码不是合法的 `JavaScript` 。这些错误可能只会作为代码运行的结果生成。代码运行可能
是 `eval` ， `Function` ， `require` 或 `vm` 的结果。这些错误经常表明了一个不完整的程序。

```
try {
  require("vm").runInThisContext("binary ! isNotOk");
} catch(err) {
  // err will be a SyntaxError
}
```

`SyntaxError` 对于创建它们的上下文来说是不可恢复的 - 它们仅可能被其他上下文捕获。

异常 vs. 错误

一个 `JavaScript` “异常”是一个无效操作或 `throw` 声明所抛出的结果的值。但是这些值不被要求必须继承于 `Error` 。所有的由 `node.js` 或 `JavaScript` 运行时抛出的异常都必须是 `Error` 实例。

一些异常在 `JavaScript` 层是无法恢复的。这些异常通常使一个进程挂掉。它们通常无法通过 `assert()` 检查，或C++层中的 `abort()` 调用。

系统错误

系统错误在程序运行时环境的响应中生成。理想情况下，它们代表了程序能够处理的操作错误。它们在系统调用级别生成：一个详尽的错误码列表和它们意义可以通过运行 `man 2 intro` 或 `man 3 errno` 在大多数 `Unices` 中获得；或在线获得。

在 `node.js` 中，系统错误表现为一个增强的 `Error` 对象 -- 不是完全的子类，而是一个有额外成员的 `error` 实例。

Class: System Error

error.syscall

一个代表了失败的系统调用的字符串。

error.errno

error.code

一个代表了错误码的字符串，通常是大写字母 `E`，可在 `man 2 intro` 命令的结果中查阅。

常见系统错误

这个列表不详尽，但是列举了许多在写 `node.js` 的过程中普遍发生的系统错误。详尽的列表可以在这里查阅： `http://man7.org/linux/man-pages/man3/errno.3.html`

EPERM: 操作不被允许

试图去执行一个需要特权的操作。

ENOENT: 指定的文件或目录不存在

通常由文件操作产生；指定的路径不存在 -- 通过指定的路径不能找到实例（文件或目录）。

EACCES: 没有权限

试图以禁止的方式去访问一个需要权限的文件。

EEXIST: 文件已存在

执行一个要求目标不存在的操作时，一个已存在文件已经是目标。

ENOTDIR: 非目录

给定的路径存在，但不是期望的目录。通常由 `fs.readdir` 产生。

EISDIR: 是目录

一个操作期望接收一个文件，但给定的路径是一个文件。

EMFILE: 系统中打开太多文件

达到了系统中允许的文件描述符的最大数量，那么下一个描述符请求，在已存在的最后一个描述符关闭之前，都不能被满足。

通常在并行打开太多文件时触发，特别是在那些将进程可用的文件描述符数量限制得很低的操作系统中（尤其是OS X）。为了改善这个限制，在同一个SHELL中运行 `ulimit -n 2048` 命令，再运行 `node.js` 进程。

EPIPE: 损坏的管道

向没有读取数据进程的管道，socket或FIFO中执行一个写操作。通常在网络和http层发生，表明需要被写入的远程流已经被关闭。

EADDRINUSE: 地址已被使用

试图给一个服务器（net，http或https）绑定一个本地地址失败，因为另一个本地系统中的服务器已经使用了那个地址。

ECONNRESET: 连接两方重置（Connection reset by peer）

连接的双方被强行关闭。通常是远程 `socket` 超时或重启的结果。通常由 `http` 和 `net` 模块产生。

ECONNREFUSED: 拒绝连接

由于目标机器积极拒绝，没有连接可以建立。通常是试图访问一个不活跃的远程主机的服务的结果。

ENOTEMPTY: 目录不为空

操作的实例要求是一个空目录，但目录不为空 -- 通常由 `fs.unlink` 产生。

ETIMEDOUT: 操作超时

因为被连接方在一段指定内未响应，连接或发送请求失败。通常由http或net产生 -- 经常是一个 被连接 `socket` 没有合适地调用 `.end()` 方法 的标志。

错误的传播和捕获

所有的 `node.js` API将无效的参数视作异常 -- 也就是说，如果传递了非法的参数，他们会立刻生成并抛出一个 `error` 作为异常，甚至是异步API也会。

同步API（像 `fs.readFileSync` ）将会抛出一个错误。抛出值的行为是将值包装入一个异常。异常可以被使用 `try { } catch(err) { }` 结果捕获。

异步API有两种错误传播机制；一种代表了单个操作（Node风格的回调函数），另一种代表了多个操作（错误事件）。

Node风格的回调函数

单个操作使用 `Node风格的回调函数` -- 一个提供给API作为参数的函数。Node风格的回调函数至少有一个参数 -- `error` -- 它可以是 `null` （如果没有错误发生）或是 `Error` 实例。例子：

```
var fs = require('fs');

fs.readFile('/some/file/that/does-not-exist', function nodeStyle
Callback(err, data) {
  console.log(err)  // Error: ENOENT
  console.log(data) // undefined / null
});

fs.readFile('/some/file/that/does-exist', function(err, data) {
  console.log(err)  // null
  console.log(data) // <Buffer: ba dd ca fe>
})
```

注意， `try { } catch(err) { }` 不能捕获异步API生成的错误。一个初学者的常见错误是尝试在Node风格的回调函数中抛出错误：

```
// THIS WILL NOT WORK:
var fs = require('fs');

try {
  fs.readFile('/some/file/that/does-not-exist', function(err, data) {
    // mistaken assumption: throwing here...
    if (err) {
      throw err;
    }
  });
} catch(err) {
  // ... will be caught here -- this is incorrect!
  console.log(err); // Error: ENOENT
}
```

这将不会正常运行！在Node风格的回调函数执行时，外围的代码 `try { } catch(err) { }` ）已经退出了。在大多数情况，在Node风格的回调函数内部抛出错误会使进程挂掉。如果启用了 `domain` ，它们可以捕获了被抛出的错误；相似的，如果给 `process.on('uncaughtException')` 添加了监听器，那么它也将会捕获错误。

错误事件

另一个提供错误的机制是 `error` 事件。这常被用在基于流或基于 `event emitter` 的API中，它们自身就代表了一系列的异步操作（每一个单一的操作都可能成功或失败）。如果在错误的源头没有添加 `error` 事件的监听器，那么 `error` 会被抛出。此时，进程会因为一个未处理的异常而挂掉，除非提供了合适的 `domains` ，或监听了 `process.on('uncaughtException')` 。

```
var net = require('net');

var connection = net.connect('localhost');

// adding an "error" event handler to a stream:
connection.on('error', function(err) {
  // if the connection is reset by the server, or if it can't
  // connect at all, or on any sort of error encountered by
  // the connection, the error will be sent here.
  console.error(err);
});

connection.pipe(process.stdout);
```

“当没有没有监听错误时会抛出错误”这个行为不仅限与 `node.js` 提供的API -- 用户创建的基于流或 `event emitters` 的API也会如此。例子：

```
var events = require('events');

var ee = new events.EventEmitter;

setImmediate(function() {
  // this will crash the process because no "error" event
  // handler has been added.
  ee.emit('error', new Error('This will crash'));
});
```

与Node风格的回调函数相同，这种方式产生的错误也不能被 `try { } catch(err) { }` 捕获 -- 它们发生时，外围的代码已经退出了。

Events

稳定度: 2 - 稳定

`node.js` 中的许多对象触发事件：一个 `net.Server` 每次被连接时触发事件，一个 `fs.readStream` 当文件打开时触发事件。所有触发事件的对象都是 `events.EventEmitter` 的实例。你可以通过 `require("events");` 来取得这个模块。

通常，事件名以驼峰字符串来命令，但是这不是严格要求的，任何字符串都是可以接受的。

为了处理触发的事件，我们将函数关联到对象上。这些函数被称为监听器。在监听器中，`this` 指向监听器所关联的 `EventEmitter` 实例。

Class: events.EventEmitter

使用 `require('events')` 来获取这个 `EventEmitter` 类。

```
var EventEmitter = require('events');
```

当一个 `EventEmitter` 实例发生了一个错误，一个典型的做法是触发一个 `error` 事件。 `error` 事件在 `node.js` 中被视为一个特殊的事件，如果没有为其添加监听器，默认的行为是打印堆栈追踪信息并推出程序。

所有的 `EventEmitter` 实例，在被添加新的监听器时，都会触发 `newListener` 事件。当有监听器被移除时，都会触发 `removeListener` 事件。

emitter.addListener(event, listener)

emitter.on(event, listener)

为指定的事件，在其监听器数组的末尾添加一个新的监听器。不会去检查这个事件是否已经被监听过。事件的多次触发会导致监听器的多次被调用。

```
server.on('connection', function (stream) {
  console.log('someone connected!');
});
```

返回一个 `emitter` ，所以可以被链式调用。

emitter.once(event, listener)

为事件添加一个 一次性 监听器。这个监听器只会在下次事件触发时被调用，之后被移除。

```
server.once('connection', function (stream) {
  console.log('Ah, we have our first user!');
});
```

返回一个 `emitter` ，所以可以被链式调用。

emitter.removeListener(event, listener)

从监听器数组中移除指定事件的一个监听器。注意：在数组中，此监听器被移除后，其之后的监听器的索引会被改变。

```
var callback = function(stream) {
  console.log('someone connected!');
};
server.on('connection', callback);
// ...
server.removeListener('connection', callback);
```

`removeListener` 一次只会从监听器数组中移除一个监听器。如果特定事件的单个的监听器被添加了多次， `removeListener` 也必须调用同样多次来移除它们。

返回一个 `emitter` ，所以可以被链式调用。

emitter.removeAllListeners([event])

移除指定事件的所有监听器。使用这个方法来 移除不是在你的代码中创建的 `emitter` （如 `socket` 和 `fs` ）的所有监听器 ，并不是一个明智的选择。

返回一个 `emitter` ，所以可以被链式调用。

emitter.setMaxListeners(n)

默认的，当一个特定事件被添加了超过10个监听器时， `EventEmitter` 会打印一个警告。这是一个对于发现内存泄露非常有用的默认警告。但是显然，并不是所有的 `emitter` 都应当被限制。这个函数可以用来增加这个上限。如果想要无限制，请设置 `0` 。

返回一个 `emitter` ，所以可以被链式调用。

emitter.getMaxListeners()

返回 `emitter` 当前的最大监听器数的值，可能是 `emitter.setMaxListeners(n)` 设置的值，或者是 `EventEmitter.defaultMaxListeners` 。

这个值对于调节最大监听器数来避免 不负责任的警告 或 最大监听器数过大，都非常有用。

```
emitter.setMaxListeners(emitter.getMaxListeners() + 1);
emitter.once('event', function () {
  // do stuff
  emitter.setMaxListeners(Math.max(emitter.getMaxListeners() - 1
, 0));
});
```

EventEmitter.defaultMaxListeners

`emitter.setMaxListeners(n)` 在实例级别设置最大监听器数。这个类属性让你可以设置所有 `EventEmitter` 的默认最大监听器数，对当前已创建的和未来创建的 `EventEmitter` 都有效。请谨慎使用它。

注意， `emitter.setMaxListeners(n)` 仍优先于 `EventEmitter.defaultMaxListeners` 。

emitter.listeners(event)

返回指定事件的监听器数组。

```
server.on('connection', function (stream) {
  console.log('someone connected!');
});
console.log(util.inspect(server.listeners('connection'))); // [
[Function] ]
```

emitter.emit(event[, arg1][, arg2][, ...])

使用提供的参数，执行每一个监听器。

如果事件有监听器，那么返回 `true` ，否则返回 `false` 。

Class Method: EventEmitter.listenerCount(emitter, event)

返回指定事件的监听器数。

Event: 'newListener'

- event String 事件名
- listener Function 事件监听器函数

这个事件在监听器被添加前触发。当这个事件被触发时，监听器还没有被添加到事件的监听器数组中。在 `newListener` 事件的回调函数中拿到事件名时，监听器还没有开始被添加到该事件。

Event: 'removeListener'

- event String 事件名
- listener Function 事件监听器函数

这个事件在监听器被移除后触发。当这个事件被触发时，监听器已经从事件的监听器数组中被移除了。

File System

稳定度: 2 - 稳定

文件I/O是由标准POSIX函数的简单包装提供的。通过 `require('fs')` 来使用这个模块。所有的方法都有异步和同步两种形式。

异步形式的方法通常在最后一个参数上接受一个回调函数。回调函数的参数则取决于不同的方法，但是第一个参数总是为异常所保留。如果操作正常结束，那么第一个参数会是 `null` 或 `undefined` 。

当同步形式的方法产生异常时，会立刻抛出。你可以使用 `try/catch` 捕获，或让它们冒泡。

下面是一个异步方法的例子：

```
var fs = require('fs');

fs.unlink('/tmp/hello', function (err) {
  if (err) throw err;
  console.log('successfully deleted /tmp/hello');
});
```

下面是一个同步方法的例子：

```
var fs = require('fs');

fs.unlinkSync('/tmp/hello');
console.log('successfully deleted /tmp/hello');
```

因为异步方法不能够保证执行顺序，所以下面的例子很容易出错：

```
fs.rename('/tmp/hello', '/tmp/world', function (err) {
  if (err) throw err;
  console.log('renamed complete');
});
fs.stat('/tmp/world', function (err, stats) {
  if (err) throw err;
  console.log('stats: ' + JSON.stringify(stats));
});
```

它需要在 `fs.rename` 后执行 `fs.stat` 。正确的执行方法应如下：

```
fs.rename('/tmp/hello', '/tmp/world', function (err) {
  if (err) throw err;
  fs.stat('/tmp/world', function (err, stats) {
    if (err) throw err;
    console.log('stats: ' + JSON.stringify(stats));
  });
});
```

在繁忙的进程中，十分推荐使用异步版本的方法。同步版本的方法会阻塞进程，直到它们完成，也就是说它们会暂停所有连接。

文件的相对路径也可以被使用，记住路径是相对于 `process.cwd()` 的。

大多数的 `fs` 函数允许你省略回调函数。如果你省略了，将会由一个默认的回调函数来重抛出（rethrows）错误。要获得原始调用地点的堆栈追踪信息，请设置 `NODE_DEBUG` 环境变量：

```
$ cat script.js
function bad() {
  require('fs').readFile('/');
}
bad();

$ env NODE_DEBUG=fs iojs script.js
fs.js:66
        throw err;
              ^
Error: EISDIR, read
    at rethrow (fs.js:61:21)
    at maybeCallback (fs.js:79:42)
    at Object.fs.readFile (fs.js:153:18)
    at bad (/path/to/script.js:2:17)
    at Object.<anonymous> (/path/to/script.js:5:1)
    <etc.>
```

fs.rename(oldPath, newPath, callback)

异步版本的 `rename(2)` 。回调函数只有一个可能的异常参数。

fs.renameSync(oldPath, newPath)

同步版本的 `rename(2)` 。返回 `undefined` 。

fs.ftruncate(fd, len, callback)

异步版本的 `ftruncate(2)` 。回调函数只有一个可能的异常参数。

fs.ftruncateSync(fd, len)

同步版本的 `ftruncate(2)` 。返回 `undefined` 。

fs.truncate(path, len, callback)

异步版本的 `truncate(2)` 。回调函数只有一个可能的异常参数。第一个参数也可以接受一个文件描述符，这样的话， `fs.ftruncate()` 会被调用。

fs.truncateSync(path, len)

同步版本的 `truncate(2)` 。返回 `undefined` 。

fs.chown(path, uid, gid, callback)

异步版本的 `chown(2)` 。回调函数只有一个可能的异常参数。

fs.chownSync(path, uid, gid)

同步版本的 `chown(2)` 。返回 `undefined` 。

fs.fchown(fd, uid, gid, callback)

异步版本的 `fchown(2)` 。回调函数只有一个可能的异常参数。

fs.fchownSync(fd, uid, gid)

同步版本的 `fchown(2)` 。返回 `undefined` 。

fs.lchown(path, uid, gid, callback)

异步版本的 `lchown(2)` 。回调函数只有一个可能的异常参数。

fs.lchownSync(path, uid, gid)

同步版本的 `lchown(2)` 。返回 `undefined` 。

fs.chmod(path, mode, callback)

异步版本的 `chmod(2)` 。回调函数只有一个可能的异常参数。

fs.chmodSync(path, mode)

同步版本的 `chmod(2)` 。返回 `undefined` 。

fs.fchmod(fd, mode, callback)

异步版本的 `fchmod(2)` 。回调函数只有一个可能的异常参数。

fs.fchmodSync(fd, mode)

同步版本的 `fchmod(2)` 。返回 `undefined` 。

fs.lchmod(path, mode, callback)

异步版本的 `lchmod(2)` 。回调函数只有一个可能的异常参数。

仅在Mac OS X中可用。

fs.lchmodSync(path, mode)

同步版本的 `lchmod(2)` 。返回 `undefined` 。

fs.stat(path, callback)

异步版本的 `stat(2)` 。回调函数有两个参数（err, stats）， `stats` 是一个 `fs.Stats` 对象。更多信息请参阅 `fs.Stats` 章节。

fs.lstat(path, callback)

异步版本的 `lstat(2)` 。回调函数有两个参数（err, stats）， `stats` 是一个 `fs.Stats` 对象。 `lstat()` 与 `stat()` 是相同的，除了 `path` 是一个符号链接，连接自己本身就是 `stat-ed` ，而不是引用一个文件。

fs.fstat(fd, callback)

异步版本的 `fstat(2)` 。回调函数有两个参数（err, stats）， `stats` 是一个 `fs.Stats` 对象。 `fstat()` 与 `stat()` 是相同的，除了将要被 `stat-ed` 的文件是通过文件描述符 `fd` 来指定的。

fs.statSync(path)

同步版本的 `stat(2)` 。返回一个 `fs.Stats` 实例。

fs.lstatSync(path)

同步版本的 `lstat(2)` 。返回一个 `fs.Stats` 实例。

fs.fstatSync(fd)

同步版本的 `fstat(2)` 。返回一个 `fs.Stats` 实例。

fs.link(srcpath, dstpath, callback)

异步版本的 `link(2)` 。回调函数只有一个可能的异常参数。

fs.linkSync(srcpath, dstpath)

同步版本的 `link(2)` 。返回 `undefined` 。

fs.symlink(destination, path[, type], callback)

异步版本的 `symlink(2)` 。回调函数只有一个可能的异常参数。 `type` 参数可以被设置为 `'dir'` ， `'file'` 或 `'junction'` （默认为 `'file'` ），并且仅在Windows平台下可用（其他平台下会被忽略）。注意Windows `junction` 点要求目标路径必须是绝对的。当使用 `'junction'` 时， `destination` 参数会被自动转换为绝对路径。

fs.symlinkSync(destination, path[, type])

同步版本的 `symlink(2)` 。返回 `undefined` 。

fs.readlink(path, callback)

异步版本的 `link(2)` 。回调函数有两个参数（err, linkString）。

fs.readlinkSync(path)

异步版本的 `readlink(2)` ，返回一个符号链接字符串值。

fs.realpath(path[, cache], callback)

异步版本的 `realpath(2)` 。回调函数有两个参数（err, resolvedPath）。可能会使用 `process.cwd` 来解析相对路径。 `cache` 是一个包含了路径映射的对象，被用来 强制进行指定的路径解析 或 避免对真实路径调用额外的 `fs.stat` 。

例子：

```
var cache = {'/etc':'/private/etc'};
fs.realpath('/etc/passwd', cache, function (err, resolvedPath) {
  if (err) throw err;
  console.log(resolvedPath);
});
```

fs.realpathSync(path[, cache])

同步版本的 `realpath(2)` ，返回一个解析出的路径。

fs.unlink(path, callback)

异步版本的 `unlink(2)` 。回调函数只有一个可能的异常参数。

fs.unlinkSync(path)

同步版本的 `unlink(2)` 。返回 `undefined` 。

fs.rmdir(path, callback)

异步版本的 `rmdir(2)` 。回调函数只有一个可能的异常参数。

fs.rmdirSync(path)

同步版本的 `rmdir(2)` 。返回 `undefined` 。

fs.mkdir(path[, mode], callback)

异步版本的 `mkdir(2)` 。回调函数只有一个可能的异常参数。 `mode` 默认为 `0o777` 。

fs.mkdirSync(path[, mode])

同步版本的 `mkdir(2)` 。返回 `undefined` 。

fs.readdir(path, callback)

异步版本的 `readdir(3)` 。读取目录内容。回调函数有两个参数（err, files），`files` 是一个目录中的文件名数组（不包括 `'.'` 和 `'..'` ）。

fs.readdirSync(path)

同步版本的 `readdir(3)` 。返回一个文件名数组（不包括 `'.'` 和 `'..'` ）。

fs.close(fd, callback)

异步版本的 `close(2)` 。回调函数只有一个可能的异常参数。

fs.closeSync(fd)

同步版本的 `close(2)` 。返回 `undefined` 。

fs.open(path, flags[, mode], callback)

异步版本的文件打开。参阅 `open(2)` 。 `flag` 可以是：

- 'r' - 以只读的方式打开文件。如果文件不存在则抛出异常。
- 'r+' - 以读写的方式打开文件。如果文件不存在则抛出异常。
- 'rs' - 同步地以只读的方式打开文件。绕过操作系统的本地文件系统缓存。

该功能主要用于打开NFS挂载的文件，因为它允许你跳过潜在的过时的本地缓存。它对I/O性能有非常大的影响，所以除非需要它，否则不应使用这个 `flag` 。

注意这个 `flag` 不会将 `fs.open()` 变为一个同步调用。因为如果你想要同步调用，你应使用 `fs.openSync()` 。

- 'rs+' - 以读写的方式打开文件，告诉操作系统同步地打开它。注意事项请参阅 `'rs'` 。
- 'w' - 以只写的方式打开文件。如果文件不存在，将会创建它。如果已存在，将会覆盖它。
- 'wx' - 类似于 `'w'` ，但是路径不存在时会失败。
- 'w+' - 以读写的方式打开文件。如果文件不存在，将会创建它。如果已存在，将会覆盖它。
- 'wx+' - 类似于 `'w+'` ，但是路径不存在时会失败。
- 'a' - 以附加的形式打开文件。如果文件不存在，将会创建它。

- 'ax' - 类似于 `'a'` ，但是路径不存在时会失败。
- 'a+' - 以读取和附加的形式打开文件。如果文件不存在，将会创建它。
- 'ax+' - 类似于 `'a+'` ，但是路径不存在时会失败。

参数 `mode` 用于设置文件模式（权限和 `sticky bits` ），但是前提是文件已被创建。它默认为 `0666` ，有可读和可写权限。

回调函数有两个参数（err, fd）。

排除标识 `'x'` （ `open(2)` 中的 `O_EXCL` 标识）保证了目录是被新创建的。在POSIX系统上，即使路径指向了一个不存在的符号链接，也会被认定为文件存在。排除标识不能保证在网络文件系统中有效。

在Linux下，无法对以追加形式打开的文件，在指定位置写入数据。内核忽略了位置参数并且总是将数据追加到文件的末尾。

fs.openSync(path, flags[, mode])

同步版本的 `fs.open()` ，返回代表文件描述符的一个整数。

fs.utimes(path, atime, mtime, callback)

更改 `path` 所指向的文件的时间戳。

fs.utimesSync(path, atime, mtime)

同步版本的 `fs.utimes()` 。返回 `undefined` 。

fs.futimes(fd, atime, mtime, callback)

更改文件描述符 `fd` 所指向的文件的时间戳。

fs.futimesSync(fd, atime, mtime)

同步版本的 `fs.futimes()` 。返回 `undefined` 。

fs.fsync(fd, callback)

异步版本的 `fsync(2)` 。回调函数只有一个可能的异常参数。

fs.fsyncSync(fd)

同步版本的 `fsync(2)` 。返回 `undefined` 。

fs.write(fd, buffer, offset, length[, position], callback)

向文件描述符 `fd` 指向的文件写入 `buffer` 。

`offset` 和 `length` 决定了 `buffer` 的哪一部分被写入文件。

`position` 指定了文件中，数据被写入的开始位置的偏移量。如果 `typeof position !== 'number'` ，那么数据将会在当前位置被写入。参阅 `pwrite(2)` 。

回调函数有三个参数（err, written, buffer）。 `written` 指出了 `buffer` 中有多少字节被写入。

注意，不等待回调函数而多次执行 `fs.write` 是不安全的。这种情况下推荐使用 `fs.createWriteStream` 。

在Linux下，无法对以追加形式打开的文件，在指定位置写入数据。内核忽略了位置参数并且总是将数据追加到文件的末尾。

fs.write(fd, data[, position[, encoding]], callback)

向文件描述符 `fd` 指向的文件写入 `data` 。如果 `data` 不是一个 `Buffer` 实例，那么其值将被强制转化为一个字符串。

`position` 指定了文件中，数据被写入的开始位置的偏移量。如果 `typeof position !== 'number'` ，那么数据将会在当前位置被写入。参阅 `pwrite(2)` 。

`encoding` 是期望的字符串编码。

回调函数有三个参数（err, written, buffer）。 `written` 指出了 `buffer` 中有多少字节被写入。注意，写入的字节与字符串字符是不同的。参阅 `Buffer.byteLength` 。

与写入 `buffer` 不同，整个字符串都必须被写入。不能指定子字符串。因为字节的偏移量可能与字符串的偏移量不相同。

注意，不等待回调函数而多次执行 `fs.write` 是不安全的。这种情况下推荐使用 `fs.createWriteStream` 。

在Linux下，无法对以追加形式打开的文件，在指定位置写入数据。内核忽略了位置参数并且总是将数据追加到文件的末尾。

fs.writeSync(fd, buffer, offset, length[, position])

fs.writeSync(fd, data[, position[, encoding]])

同步版本的 `fs.write()` 。返回被写入的字节数。

fs.read(fd, buffer, offset, length, position, callback)

从文件描述符 `fd` 指向的文件读取数据。

`buffer` 是数据将要被写入的缓冲区。

`offset` 是开始向 `buffer` 写入数据的缓冲区偏移量。

`length` 是一个指定了读取字节数的整数。

`position` 是一个指定了从文件的何处开始读取数据的整数。如果 `position` 是 `null` ，数据将会从当前位置开始读取。

回调函数有三个参数（err, bytesRead, buffer）。

fs.readSync(fd, buffer, offset, length, position)

同步版本的 `fs.read` 。返回读取字节的个数。

fs.readFile(filename[, options], callback)

- filename String
- **options Object | String**
  - encoding String | Null 默认为 `null`
  - flag String 默认为 `'r'`
- callback Function

异步得读取文件的所有内容。例子：

```
fs.readFile('/etc/passwd', function (err, data) {
  if (err) throw err;
  console.log(data);
});
```

回调函数有两个参数（err, data），`data` 是文件的内容。

如果没有指定编码，那么将会返回源 `buffer` 。

如果 `options` 是一个字符串，那么它将指定编码，例子：

```
fs.readFile('/etc/passwd', 'utf8', callback);
```

fs.readFileSync(filename[, options])

同步版本的 `fs.readFile` 。返回文件的内容。

如果指定了编码那么将会返回字符串。否则返回 `buffer` 。

fs.writeFile(filename, data[, options], callback)

- filename String
- data String | Buffer
- **options Object | String**
    - encoding String | Null 默认为 `'utf8'`
    - mode Number 默认为 `0o666`
    - flag String 默认为 `'w'`
- callback Function

异步地向文件写入数据，如果文件已经存在，那么会覆盖它。`data` 可以是一个字符串或一个 `buffer` 。

如果数据时一个 `buffer` 那么编码会被忽略。编码默认为 `'utf8'` 。

例子：

```
fs.writeFile('message.txt', 'Hello node.js', function (err) {
  if (err) throw err;
  console.log('It\'s saved!');
});
```

如果 `options` 是一个字符串，那么它将指定编码，例子：

```
fs.writeFile('message.txt', 'Hello node.js', 'utf8', callback);
```

fs.writeFileSync(filename, data[, options])

同步版本的 `fs.writeFile` 。返回 `undefined` 。

fs.appendFile(filename, data[, options], callback)

- filename String
- data String | Buffer
- **options Object | String**
  - encoding String | Null 默认为 `'utf8'`
  - mode Number 默认为 `0o666`
  - flag String 默认为 `'a'`
- callback Function

异步地向文件追加数据，如果文件不存在将会创建它。 `data` 可以是一个字符串或一个 `buffer` 。

例子：

```
fs.appendFile('message.txt', 'data to append', function (err) {
  if (err) throw err;
  console.log('The "data to append" was appended to file!');
});
```

如果 `options` 是一个字符串，那么它将指定编码，例子：

```
fs.appendFile('message.txt', 'data to append', 'utf8', callback)
;
```

fs.appendFileSync(filename, data[, options])

同步版本的 `fs.appendFile` 。返回 `undefined` 。

fs.watchFile(filename[, options], listener)

监视文件变化。回调函数 `listener` 会在文件每一次被访问时调用。

第二参数是可选的。如果 `options` 被提供，那么它必须是一个含有两个成员 `persistent` 和 `interval` 的对象。 `persistent` 表明了进程是否在文件被监视时继续执行。 `interval` 表明了文件被轮询的间隔（毫秒）。默认是 `{ persistent: true, interval: 5007 }` 。

`listener` 有两个参数，当前状态对象和先前状态对象：

```
fs.watchFile('message.text', function (curr, prev) {
  console.log('the current mtime is: ' + curr.mtime);
  console.log('the previous mtime was: ' + prev.mtime);
});
```

这两个状态对象都是 `fs.Stat` 实例。

如果你想要在文件被修改时被通知，而不仅仅是在被访问时，你需要比较 `curr.mtime` 和 `prev.mtime` 。

注意： `fs.watch` 比 `fs.watchFile` 和 `fs.unwatchFile` 更高效。当可能时，请使用 `fs.watch` 替代它们。

fs.unwatchFile(filename[, listener])

停止监视 `filename` 的变化。如果指定了 `listener` ，那么仅仅会移除指定的 `listener` 。否则所有的监听器都会被移除，并且停止继续监视文件。

对一个没有被监视的文件调用 `fs.unwatchFile()` 将不会发生任何事，而不是报错。

注意：`fs.watch` 比 `fs.watchFile` 和 `fs.unwatchFile` 更高效。当可能时，请使用 `fs.watch` 替代它们。

fs.watch(filename[, options][, listener])

监视 `filename` 的变化，`filename` 指向的可以是文件也可以是目录。返回一个 `fs.FSWatcher` 对象。

第二个参数是可选的。`options` 必须是一个对象。支持的布尔值属性是 `persistent` 和 `recursive` 。`persistent` 表明了进程是否在文件被监视时继续执行。`recursive` 表明了是否子目录也需要被监视，或仅仅监视当前目录。这只在支持的平台（参阅下方 `警告` ）下传递一个目录时有效。

默认是 `{ persistent: true, recursive: false }` 。

`listener` 回调函数有两个参数（event, filename）。`event` 是 `'rename'` 或 `'change'` ，`filename` 是触发事件的文件名。

警告

`fs.watch` API 不是在所有平台下都表现一致的，并且在一些情况下是不可用的。

`recursive` 选项目前只支持OS X。只有 `FSEvents` 支持这种类型的文件监控，所有其他平台并不会很快都被支持。

可用性

这个特性依赖于底层操作系统提供的文件变化提示。

- 在Linux系统下，它使用 `inotify` 。
- 在BSD系统下，它使用 `kqueue` 。
- 在OS X下，对于文件它使用 `kqueue` ，对于目录它使用 `FSEvents` 。
- 在SunOS系统（包括 `Solaris` 和 `SmartOS` ）下，它使用事件端口（event ports）。
- 在Windows系统下，这个特性依赖于 `ReadDirectoryChangesW` 。

如果由于一些原因，底层功能不可用，那么 `fs.watch` 的功能也将不可用。例如，在网络文件系统（NFS，SMB等）中监视文件或目录变化，往往结果不可靠或完全不可用。

你仍可以使用 `fs.watchFile` ，它使用了状态轮询。但是性能更差且可靠性更低。

Filename 参数

回调函数中提供的 `filename` 参数不是在所有平台上都支持的（目前只支持Linux和Windows）。即使是在支持的平台上， `filename` 也不是总会被提供。因此，不要假设 `filename` 参数总会在回调函数中被提供，需要有一些检测它是否为 `null` 的逻辑。

```
fs.watch('somedir', function (event, filename) {
  console.log('event is: ' + event);
  if (filename) {
    console.log('filename provided: ' + filename);
  } else {
    console.log('filename not provided');
  }
});
```

fs.exists(path, callback)

`fs.exists()` 已被弃用。请使用 `fs.stat` 或 `fs.access` 替代。

检查文件系统来测试提供的路径是否存在。然后在回调函数的参数中提供结果 `true` 或 `false` ：

```
fs.exists('/etc/passwd', function (exists) {
  util.debug(exists ? "it's there" : "no passwd!");
});
```

`fs.exists()` 是一个不符合潮流的函数，并且仅因一些历史原因所以仍然错在。在你的代码中，不应有任何原因要继续使用它。

特别的，在打开文件前检查文件是否存在 是一种反模式。因为竞态条件所以让你的代码十分脆弱：其他进程可能 `fs.exists()` 和 `fs.open()` 之间删除文件。所以仅仅就去打开一个文件，并且当它不存在时处理错误。

fs.existsSync(path)

同步版本的 `fs.exists` 。当文件存在，返回 `true` ，否则返回 `false` 。

`fs.existsSync()` 已被弃用。请使用 `fs.statSync` 或 `fs.accessSync` 替代。

fs.access(path[, mode], callback)

对于指定的路径，检测用户的权限。 `mode` 是一个可选的整数，指定了要被执行的可访问性检查。以下是 `mode` 的一些可用的常量。可以通过“或”运算符（|）连接两个或以上的值。

- fs.F_OK - 文件对于当前进程可见。这对于检查文件是否存在很有用，但是不提供任何 `rwx` 权限信息。这是默认值。
- fs.R_OK - 文件对于当前进程可读。
- fs.W_OK - 文件对于当前进程可写。
- fs.X_OK - 文件对于当前进程可执行。这在Windows上无效（将会表现得像 `fs.F_OK` 一样）。

最后一个参数 `callback` ，是一个包含了潜在错误参数的回调函数。如果任何一个可访问检查失败了，错误参数就会被提供。以下是一个在当前进程中检查 `/etc/passwd` 可读性和可写性的例子。

```
fs.access('/etc/passwd', fs.R_OK | fs.W_OK, function(err) {
  util.debug(err ? 'no access!' : 'can read/write');
});
```

fs.accessSync(path[, mode])

同步版本的 `fs.access` 。如果任何一个可访问性检查失败了，它会抛出异常。否则什么都不做。

Class: fs.Stats

由 `fs.stat()` ， `fs.lstat()` ， `fs.lstat()` 和它们的同步版本函数所返回的对象。

- stats.isFile()
- stats.isDirectory()
- stats.isBlockDevice()
- stats.isCharacterDevice()

- stats.isSymbolicLink()（仅在调用 `fs.lstat()` 时有效）
- stats.isFIFO()
- stats.isSocket()

对于一个普通的文件， `util.inspect(stats)` 可能会返回：

```
{ dev: 2114,
  ino: 48064969,
  mode: 33188,
  nlink: 1,
  uid: 85,
  gid: 100,
  rdev: 0,
  size: 527,
  blksize: 4096,
  blocks: 8,
  atime: Mon, 10 Oct 2011 23:24:11 GMT,
  mtime: Mon, 10 Oct 2011 23:24:11 GMT,
  ctime: Mon, 10 Oct 2011 23:24:11 GMT,
  birthtime: Mon, 10 Oct 2011 23:24:11 GMT }
```

请注意， `atime` ， `mtime` ， `birthtime` 和 `ctime` 都是 `Date` 对象实例，并且你可以通过合适的方法来比较它们的值。普遍的使用方式是，调用 `getTime()` 来获取unix时间戳并且这个整数可以被用来进行任何比较。但是还有一些可以展示模糊信息的方法。更多的详细信息请参阅 `MDN JavaScript Reference` 页。

Stat 时间值

`stat` 对象中的各个时间有如下语义：

- atime "访问时间" - 文件数据最后一次被访问时的时间。
  由 `mknod(2)` ， `utimes(2)` 和 `read(2)` 系统调用改变。
- mtime "修改时间" - 文件数据最后一次被修改的时间。
  由 `mknod(2)` ， `utimes(2)` 和 `write(2)` 系统调用改变。
- ctime "改变时间" - 文件状态最后一次被改变（索引节点改变）的时间。
  由 `chmod(2)` ， `chown(2)` ， `link(2)` ， `mknod(2)` ， `rename(2)` ， `unlink(2)` ， `utimes(2)` ， `read(2)` 和 `write(2)` 系统调用改变。
- birthtime "创建时间" - 文件的创建时间。在文件被创建时设置。在创建时间不

可用的的文件系统上，这个值可能会被 `ctime` 或是 `1970-01-01T00:00Z` （unix时间戳0）填充。在Darwin或其他FreeBSD系统变体上，如果使用 `utimes(2)` 系统调用设置 `atime` 为一个比当前 `birthtime` 更早的时间， `birthtime` 也会被这样填充。

在 `node.js` v1.0 和 Node v0.12 前，Windows系统中 `ctime` 持有了 `birthtime` 值。但是在 v0.12 里， `ctime` 不再是“创建时间”。在Unix系统中，它从来都不是。

fs.createReadStream(path[, options])

返回一个新的可读流对象（参阅 `Readable Stream` ）。

`options` 是一个有以下默认值的对象或字符串：

```
{ flags: 'r',
  encoding: null,
  fd: null,
  mode: 0o666,
  autoClose: true
}
```

`options` 可以包含 `start` 和 `end` 值来读取指定范围的文件数据。 `start` 和 `end` 这两个位置本身，也都是被包括的，并且 `start` 以 `0` 开始。编码可以是 `'utf8'` ， `'ascii'` 或 `'base64'` 。

如果指定了 `fd` ，可读流将会忽略 `path` 参数并且将会使用指定的文件描述符。这意味 `open` 事件不再会触发。

如果 `autoClose` 为 `false` ，那么文件描述符将不会被关闭，甚至是有错误发生时。关闭它将是你的责任，并且要确保没有文件描述符泄漏。如果 `autoClose` 为 `true` （默认），那么在发生错误时，或到达文件描述末端时，它会被自动关闭。

从一个100字节的文件中读取最后10字节数据的例子：

```
fs.createReadStream('sample.txt', {start: 90, end: 99});
```

如果 `options` 是一个字符串，那么它表示指定的编码。

Class: fs.ReadStream

`ReadStream` 是一个可读流。

Event: 'open'

- fd Integer 被可读流使用的文件描述符

当可读流文件被打开时触发。

fs.createWriteStream(path[, options])

返回一个新的可写流对象（参阅 `Writable Stream` ）。

`options` 是一个有以下默认值的对象或字符串：

```
{ flags: 'w',
  encoding: null,
  fd: null,
  mode: 0o666 }
```

`options` 可以包含一个 `start` 选项来允许从指定位置开始写入数据。修改一个文件而不是替换它，需要一个 `r+` 标识，而不是默认的 `w` 。编码可以是 `'utf8'` ， `'ascii'` ， `'binary'` 或 `'base64'` 。

与上文的 `ReadStream` 类似，如果指定了 `fd` ，可写流会忽略 `path` 参数，并且使用指定的文件描述符。这意味 `open` 事件不再会触发。

如果 `options` 是一个字符串，那么它表示指定的编码。

Class: fs.WriteStream

`WriteStream` 是一个可写流。

Event: 'open'

- fd Integer `WriteStream` 使用的文件描述符

当可写流文件被打开时触发。

file.bytesWritten

至今为止写入的字节数。不包括仍在写入队列中的数据。

Class: fs.FSWatcher

由 `fs.watch()` 返回的对象。

watcher.close()

停止在指定的 `fs.FSWatcher` 上监视文件变化。

Event: 'change'

- event String 文件的改变类型
- filename String The filename that changed (if relevant/available)被改变的文件（如果有意义/可用的话）

当被监视的目录或文件发生了改变时触发。详情参阅 `fs.watch` 。

Event: 'error'

- error Error object

当错误发生时触发。

Global Objects

这些对象是所有模块都可用的。其中的一些对象不是真正的在全局作用域内，而是在模块作用域内 - 它将会在文档中被指出。

global

- {Object} 全局命名空间对象。

在浏览器，顶级作用域是全局作用域。这意味着在浏览器的全局作用域中，你创建了一个对象那么就是定义了一个全局对象。在 `node.js` 中是不同的，顶级作用域不是全局作用域，在 `node.js` 的模块中创建的对象只属于那个模块。

process

- {Object}

进程对象。参阅 `process` 章节。

console

- {Object}

被用来向 `stdout` 和 `stderr` 打印信息。参阅 `console` 章节。

Class: Buffer

- {Function}

被用来处理二进制数据。参阅 `buffer` 章节。

require()

- {Function}

用来引入模块。参阅 `Modules` 章节。 `require` 实际上不是全局的，而是每个模块本地的。

require.resolve()

使用内部 `require()` 机制来查找模块位置，但是只返回被解析的模块路径，而不是加载模块。

require.cache

- Object

当模块被引入时，模块在这个对象中被缓存。通过删除这个对象的键值，下一次引入会重新加载模块。

require.extensions

稳定度: 0 - 弃用

- Object

指示 `require` 方法如何处理特定的文件扩展名。

将扩展名为 `.sjs` 的文件当做 `.js` 文件处理：

```
require.extensions['.sjs'] = require.extensions['.js'];
```

在被弃用之前，这个列表被用于按需编译非 `JavaScript` 模块并加载入 `node.js` 。但是，在实践中，有更好地方法来实现这个功能，如使用其他的 `node.js` 程序来加载模块，或在预编译为 `JavaScript` 。

由于模块系统的API已被锁定，这个特性可能永远不会被去处。但是它可能有细微的bug和额外的复杂性，所以最好不要再使用它。

__filename

- {String}

当前被指定的代码的文件名。它被解析为绝对路径。对于主程序，它可能与命令行中使用的文件路径是不同的。在模块内这个值是该模块文件的路径。

例子：在 `/Users/mjr` 目录中执行 `iojs example.js`

```
console.log(__filename);
// /Users/mjr/example.js
```

`__filename` 实际上不是全局的，而是每个模块本地的。

__dirname

- {String}

当前执行脚本所在的目录名。

例子：在 `/Users/mjr` 目录中执行 `iojs example.js`

```
console.log(__dirname);
// /Users/mjr
```

`__dirname` 实际上不是全局的，而是每个模块本地的。

module

- {Object}

当前模块的一个引用。特别的，`module.exports` 被用来指定模块需要对外暴露的东西，这些东西可以通过 `require()` 取得。

`module` 实际上不是全局的，而是每个模块本地的。

更多信息请参阅模块系统文档。

exports

`module.exports` 的一个快捷引用。对于何时使用 `exports` ，何时使用 `module.exports` ，请参阅模块系统文档。

`exports` 实际上不是全局的，而是每个模块本地的。

更多信息请参阅模块系统文档。

更多信息请参阅 `module` 章节。

setTimeout(cb, ms)

在至少 `ms` 毫秒后，执行回调函数 `cb` 。实际的延时依赖于外部因素，如操作系统的定时器粒度和系统负载。

超时时间必须在1到2,147,483,647之间。如果超过了这个范围，它会被重置为1毫秒。换句话说，定时器的跨度不可以超过24.8天。

返回一个代表此定时器的句柄值。

clearTimeout(t)

停止一个之前通过 `setTimeout()` 创建的定时器。它的回调函数将不会执行。

setInterval(cb, ms)

以 `ms` 毫秒的间隔，重复地执行回调函数 `cb` 。实际的间隔可能会有浮动，这取决于外部因素，如操作系统的定时器粒度和系统负载。它永远不会比 `ms` 短只会比它长。

间隔值必须在1到2,147,483,647之间。如果超过了这个范围，它会被重置为1毫秒。换句话说，定时器的跨度不可以超过24.8天。

返回一个代表此定时器的句柄值。

clearInterval(t)

停止一个之前通过 `setInterval()` 创建的定时器。它的回调函数将不会执行。

定时器函数都是全局变量。参阅定时器章节。

HTTP

稳定度: 2 - 稳定

你必须通过 `require('http')` 来使用HTTP服务器和客户端。

`node.js` 中的HTTP接口被设置来支持许多HTTP协议里原本用起来很困难的特性。特别是大且成块的有编码的消息。这个接口从不缓冲整个请求或响应。用户可以对它们使用流。

HTTP消息头可能是一个类似于以下例子的对象：

```
{ 'content-length': '123',
  'content-type': 'text/plain',
  'connection': 'keep-alive',
  'host': 'mysite.com',
  'accept': '*/*' }
```

键是小写的。值没有被修改。

为了全方位的支持所有的HTTP应用。 `node.js` 的HTTP API是非常底层的。它只处理流以及解释消息。它将消息解释为消息头和消息体，但是不解释实际的消息头和消息体。

被定义的消息头允许以多个 `,` 字符分割，除了 `set-cookie` 和 `cookie` 头，因为它们表示值得数组。如 `content-length` 这样只有单个值的头被直接将解析，并且成为解析后对象的一个单值。

收到的原始消息头会被保留在 `rawHeaders` 属性中，它是一个形式如 `[key, value, key2, value2, ...]` 的数组。例如，之前的消息头可以有如下的 `rawHeaders` ：

```
[ 'ConTent-Length', '123456',
  'content-LENGTH', '123',
  'content-type', 'text/plain',
  'CONNECTION', 'keep-alive',
  'Host', 'mysite.com',
  'accepT', '*/*' ]
```

http.METHODS

- Array

一个被解析器所支持的HTTP方法的列表。

http.STATUS_CODES

- Object

一个所有标准HTTP响应状态码的集合，以及它们的简短描述。例如， `http.STATUS_CODES[404] === 'Not Found'` 。

http.createServer([requestListener])

- 返回一个新的 `http.Server` 实例

`requestListener` 是一个会被自动添加为 `request` 事件监听器的函数。

http.createClient([port][, host])

这个函数已经被启用。请使用 `http.request()` 替代。构造一个新的HTTP客户端。 `port` 和 `host` 指定了需要连接的目标服务器。

Class: http.Server

这是一个具有以下事件的 `EventEmitter` ：

Event: 'request'

- function (request, response) { }

当有请求来到时触发。注意每一个连接可能有多个请求（在长连接的情况下）。请求是一个 `http.IncomingMessage` 实例，响应是一个 `http.ServerResponse` 实例。

Event: 'connection'

- function (socket) { }

当一个新的TCP流建立时触发。 `socket` 是一个 `net.Socket` 类型的实例。用户通常不会接触这个事件。特别的，因为协议解释器绑定它的方式， `socket` 将不会触发 `readable` 事件。这个 `socket` 可以由 `request.connection` 得到。

Event: 'close'

- function () { }

当服务器关闭时触发。

Event: 'checkContinue'

- function (request, response) { }

当每次收到一个HTTP `Expect: 100-continue` 请求时触发。如果不监听这个事件，那么服务器会酌情自动响应一个 `100 Continue` 。

处理该事件时，如果客户端可以继续发送请求主体则调用 `response.writeContinue()` ，如果不能则生成合适的HTTP响应（如 `400 Bad Request` ）。

注意，当这个事件被触发并且被处理， `request` 事件则不会再触发。

Event: 'connect'

- function (request, socket, head) { }

每当客户端发起一个http `CONNECT` 请求时触发。如果这个事件没有被监听，那么客户端发起http `CONNECT` 的连接会被关闭。

- `request` 是一个http请求参数，它也被包含在 `request` 事件中。
- `socket` 是一个服务器和客户端间的网络套接字。
- `head` 是一个 `Buffer` 实例，隧道流中的第一个报文，该参数可能为空

在这个事件被触发后，请求的 `socket` 将不会有 `data` 事件的监听器，意味着你将要绑定一个 `data` 事件的监听器来处理这个 `socket` 中发往服务器的数据。

Event: 'upgrade'

- function (request, socket, head) { }

每当客户端发起一个http `upgrade` 请求时触发。如果这个事件没有被监听，那么客户端发起 `upgrade` 的连接会被关闭。

- `request` 是一个http请求参数，它也被包含在 `request` 事件中。
- `socket` 是一个服务器和客户端间的网络套接字。
- `head` 是一个 `Buffer` 实例，升级后流中的第一个报文，该参数可能为空

在这个事件被触发后，请求的 `socket` 将不会有 `data` 事件的监听器，意味着你将要绑定一个 `data` 事件的监听器来处理这个 `socket` 中发往服务器的数据。

Event: 'clientError'

- function (exception, socket) { }

如果一个客户端连接发生了错误，这个事件将会被触发。

`socket` 是一个错误来源的 `net.Socket` 对象。

server.listen(port[, hostname][, backlog][, callback])

从指定的端口和主机名开始接收连接。如果 `hostname` 被忽略，那么如果IPv6可用，服务器将接受任意IPv6地址（::），否则为任何IPv4地址（0.0.0.）。 `port` 为 `0` 将会设置一个随机端口。

如果要监听一个unix `socket` ，请提供一个文件名而不是端口和主机名。

`backlog` 是连接等待队列的最大长度。它的实际长度将有你操作系统的 `sysctl` 设置（如linux中的 `tcp_max_syn_backlog` 和 `somaxconn` ）决定。默认值为 `511` （不是 `512` ）。

这个函数式异步的。最后一个 `callback` 参数将会添加至 `listening` 事件的监听器。参阅 `net.Server.listen(port)` 。

server.listen(path[, callback])

通过给定的 `path` ，开启一个监听连接的 UNIX `socket` 服务器。

这个函数式异步的。最后一个 `callback` 参数将会添加至 `listening` 事件的监听器。参阅 `net.Server.listen(path)` 。

server.listen(handle[, callback])

- handle Object
- callback Function

`handle` 对象是既可以是一个server可以是一个 `socket` （或者任意以下划线开头的成员 `_handle` ），或一个 `{fd: <n>}` 对象。

这将使得服务器使用指定句柄接受连接，但它假设文件描述符或句柄已经被绑定至指定的端口或域名 `socket` 。

在Windows下不支持监听一个文件描述符。

这个函数式异步的。最后一个 `callback` 参数将会添加至 `listening` 事件的监听器。参阅 `net.Server.listen()` 。

server.close([callback])

让服务器停止接收新的连接。参阅 `net.Server.close()` 。

server.maxHeadersCount

限制最大请求头数量，默认为 `1000` 。如果设置为 `0` ，则代表无限制。

server.setTimeout(msecs, callback)

- msecs Number
- callback Function

设置 `socket` 的超时值，并且如果超时，会在服务器对象上触发一个 `timeout` 事件，并且将传递 `socket` 作为参数。

如果在服务器对象时又一个 `timeout` 事件监听器，那么它将会被调用，而超时的 `socket` 将会被作为参数。

默认的，服务器的超时值是两分钟，并且如果超时， `socket` 会被自动销毁。但是，如果你给 `timeout` 事件传递了回调函数，那么你必须为要亲自处理 `socket` 超时。

返回一个 `server` 对象。

server.timeout

- Number 默认为 `120000` （两分钟）

一个 `socket` 被判定为超时之前的毫秒数。

注意， `socket` 的超时逻辑在连接时被设定，所以改变它的值仅影响之后到达服务器的连接，而不是所有的连接。

设置为 `0` 将会为连接禁用所有的自动超时行为。

Class: http.ServerResponse

这个对象由HTTP服务器内部创建，而不是由用户。它会被传递给 `request` 事件监听器的第二个参数。

这个对象实现了 `Writable` 流接口。它是一个具有以下事件的 `EventEmitter` ：

Event: 'close'

- function () { }

表明底层的连接在 `response.end()` 被调用或能够冲刷前被关闭。

Event: 'finish'

- function () { }

当响应被设置时触发。更明确地说，这个事件在当响应头的最后一段和响应体为了网络传输而交给操作系统时触发。它并不表明客户端已经收到了任何信息。

这个事件之后， `response` 对象不会再触发任何事件。

response.writeContinue()

给客户端传递一个 `HTTP/1.1 100 Continue` 信息，表明请求体必须被传递。参阅服务器的 `checkContinue` 事件。

response.writeHead(statusCode[, statusMessage][, headers])

为请求设置一个响应头。 `statusCode` 是一个三位的HTTP状态码，如 `404` 。最后一个参数 `headers` ，是响应头。第二个参数 `statusMessage` 是可选的，表示状态码的一个可读信息。

例子：

```
var body = 'hello world';
response.writeHead(200, {
  'Content-Length': body.length,
  'Content-Type': 'text/plain' });
```

这个方法对于一个信息只能调用一次，并且它必须在 `response.end()` 之前被调用。

如果你在调用这个方法前调用了 `response.write()` 或 `response.end()` ，将会调用这个函数，并且一个 `implicit/mutable` 头会被计算使用。

注意， `Content-Length` 是以字节计，而不是以字符计。上面例子能正常运行时因为字符串 `'hello world'` 仅包含单字节字符。如果响应体包含了多字节编码的字符，那么必须通过指定的编码来调用 `Buffer.byteLength()` 来确定字节数。并且 `node.js` 不会检查 `Content-Length` 与响应体的字节数是否相等。

response.setTimeout(msecs, callback)

- msecs Number
- callback Function

设置 `socket` 的超时值（毫秒），如果传递了回调函数，那么它将被添加至 `response` 对象的 `timeout` 事件的监听器。

如果没有为 `request` ， `request` 或服务器添加 `timeout` 监听器。那么 `socket` 会在超时时销毁。如果你为 `request` ， `request` 或服务器添加了 `timeout` 监听器，那么你必须为要亲自处理 `socket` 超时。

返回一个 `response` 对象。

response.statusCode

当使用隐式响应头（不明确调用 `response.writeHead()` ）时，这个属性控制了发送给客户端的状态码，在当响应头被冲刷时。

例子：

```
response.statusCode = 404;
```

在响应头发送给客户端之后，这个属性表明了被发送的状态码。

response.statusMessage

当使用隐式响应头（不明确调用 `response.writeHead()` ）时，这个属性控制了发送给客户端的状态信息，在当响应头被冲刷时。当它没有被指定（ `undefined` ）时，将会使用标准HTTP状态码信息。

例子：

```
response.statusMessage = 'Not found';
```

在响应头发送给客户端之后，这个属性表明了被发送的状态信息。

response.setHeader(name, value)

为一个隐式的响应头设置一个单独的头内容。如果这个头已存在，那么将会被覆盖。当你需要发送一个同名多值的头内容时请使用一个字符串数组。

例子：

```
response.setHeader("Content-Type", "text/html");
//or

response.setHeader("Set-Cookie", ["type=ninja", "language=javasc
ript"]);
```

response.headersSent

布尔值（只读）。如果响应头被发送则为 `true` ，反之为 `false` 。

response.sendDate

当为 `true` 时，当响应头中没有 `Date` 值时会被自动设置。默认为 `true` 。

这个值只会为了测试目的才会被禁用。HTTP协议要求响应头中有 `Date` 值。

response.getHeader(name)

读取已经被排队但还未发送给客户端的响应头。注意 `name` 是大小写敏感的。这个函数只能在响应头被隐式冲刷前被调用。

例子：

```
var contentType = response.getHeader('content-type');
```

response.removeHeader(name)

取消一个在队列中等待隐式发送的头。

例子：

```
response.removeHeader("Content-Encoding");
```

response.write(chunk[, encoding][, callback])

如果这个方法被调用并且 `response.writeHead()` 没有备调用，那么它将转换到隐式响应头模式，并且刷新隐式响应头。

这个方法传递一个数据块的响应体。这个方法可能被调用多次来保证连续的提供响应体。

数据块可以是一个字符串或一个 `buffer` 。如果数据块是一个字符串，那么第二个参数是它的编码。默认是UTF-8.最后一个回调函数参数会在数据块被冲刷后触发。注意：这是一个底层的HTTP报文，高级的多部分报文编码无法使用。

第一次调用 `response.write()` 时，它会传递缓存的头信息以及第一个报文给客户端。第二次调用时， `node.js` 假设你将发送数据流，然后分别发送。这意味着响应式缓冲到第一个报文的数据块中。

如果整个数据都成功得冲刷至内核缓冲，则放回 `true` 。如果用户内存中有部分或全部的数据在队列中，那么返回 `false` 。 `drain` 事件将会在缓冲再次释放时触发。

response.addTrailers(headers)

这个方法添加HTTP尾随头（一个在消息最后的头）给响应。

只有当数据编码被用于响应时尾随才会触发。如果不是（如请求是 `HTTP/1.0` ），它们将被安静地丢弃。

注意，如果你要触发尾随消息，HTTP要求传递一个包含报文头场列表的尾随头：

```
response.writeHead(200, { 'Content-Type': 'text/plain',
                          'Trailer': 'Content-MD5' });
response.write(fileData);
response.addTrailers({'Content-MD5': "7895bf4b8828b55ceaf47747b4
bca667"});
response.end();
```

response.end([data][, encoding][, callback])

这个方法告知服务器所有的响应头和响应体都已经发送；服务器会认为这个消息完成了。这个方法必须在每次响应完成后被调用。

如果指定了 `data` ，就相当于调用了 `response.write(data, encoding)` 之后再调用 `response.end(callback)` 。

如果指定了回调函数，那么它将在响应流结束后触发。

http.request(options[, callback])

`node.js` 为每个服务器维护了几个连接，用来产生HTTP请求。这函数允许你透明地发送请求。

`options` 参数可以是一个对象或一个字符串，如果 `options` 是一个字符串，它将自动得被 `url.parse()` 翻译。

Options:

- host: 一个将要向其发送请求的服务器域名或IP地址。默认为 `localhost` 。
- hostname: `host` 的别名。为了支持 `url.parse()` 的话， `hostname` 比 `host` 更好些。
- family: 解析 `host` 和 `hostname` 时的IP地址协议族。合法值是 `4` 和 `6` 。当没有指定时，将都被使用。
- port: 远程服务器端口。默认为 `80` 。
- localAddress: 用于绑定网络连接的本地端口。
- socketPath: Unix域 `socket` （使用 `host:port` 或 `socketPath` ）。
- method: 指定HTTP请求方法的字符串。默认为 `GET` 。
- path: 请求路径。默认为 `/` 。如果有查询字符串，则需要包含。例如'/index.html?page=12'。请求路径包含非法字符时抛出异常。目前，只否决空格，不过在未来可能改变。
- headers: 一个包含请求头的对象。
- auth: 用于计算认证头的基本认证，即 `'user:password'` 。
- **agent**: 控制 `agent` 行为。当使用一个代理时，请求将默认为 `Connection: keep-alive` 。可能值有：
  - undefined (默认): 在这个主机和端口上使用全局`agent。
  - Agent object: 在 `agent` 中显示使用passed。
  - false: 跳出 `agent` 的连接池。默认请求为 `Connection: close` 。

可选的回调函数将会被添加为 `response` 事件的“一次性”监听器（one time listener）。

`http.request()` 返回一个 `http.ClientRequest` 类的实例。这个 `ClientRequest` 实例是一个可写流。如果你需要使用 `POST` 请求上传一个文件，那么就将之写入这个 `ClientRequest` 对象。

例子：

```
var postData = querystring.stringify({
  'msg' : 'Hello World!'
});

var options = {
  hostname: 'www.google.com',
  port: 80,
  path: '/upload',
  method: 'POST',
  headers: {
    'Content-Type': 'application/x-www-form-urlencoded',
    'Content-Length': postData.length
  }
};

var req = http.request(options, function(res) {
  console.log('STATUS: ' + res.statusCode);
  console.log('HEADERS: ' + JSON.stringify(res.headers));
  res.setEncoding('utf8');
  res.on('data', function (chunk) {
    console.log('BODY: ' + chunk);
  });
  res.on('end', function() {
    console.log('No more data in response.')
  })
});

req.on('error', function(e) {
  console.log('problem with request: ' + e.message);
});

// write data to request body
req.write(postData);
req.end();
```

注意，在例子中调用了 `req.end()` 。使用 `http.request()` 时必须调用 `req.end()` 来表明你已经完成了请求（即使没有数据要被写入请求体）。

如果有一个错误在请求时发生（如DNS解析，TCP级别错误或实际的HTTP解析错误），一个 `error` 事件将会在返回对象上触发。

下面有一些特殊的需要主要的请求头：

- 发送 `'Connection: keep-alive'` 会告知 `node.js` 保持连直到下一个请求发送。
- 发送 `'Content-length'` 头会禁用默认的数据块编码。
- 发送 `'Expect'` 头将会立刻发送一个请求头。通常，当发送 `'Expect: 100-continue'` 时，你需要同时设置一个超时和监听后续的时间。参阅RFC2616的8.2.3章节来获取更多信息。
- 发送一个授权头将会覆盖使用 `auth` 选项来进行基本授权。

http.get(options[, callback])

由于大多数请求是没有请求体的 `GET` 请求。 `node.js` 提供了这个简便的方法。这个方法和 `http.request()` 方法的唯一区别是它设置请求方法为 `GET` 且自动调用 `req.end()` 。

例子：

```
http.get("http://www.google.com/index.html", function(res) {
  console.log("Got response: " + res.statusCode);
}).on('error', function(e) {
  console.log("Got error: " + e.message);
});
```

Class: http.Agent

HTTP Agent是用来把HTTP客户端请求中的 `socket` 做成池。

HTTP Agent 也把客户端的请求默认为使用 `Connection:keep-alive` 。如果没有HTTP请求正在等待成为空闲的套接字的话，那么套接字将关闭。这意味着 `node.js` 的资源池在负载的情况下对 `keep-alive` 有利，但是仍然不需要开发人员使用KeepAlive来手动关闭HTTP客户端。

如果你选择使用 `HTTP KeepAlive` ，那么你可以创建一个标志设为 `true` 的Agent对象（见下面的构造函数选项）。然后，Agent将会在资源池中保持未被使用的套接字，用于未来使用。它们将会被显式标记，以便于不保持 `node.js` 进程的运行。但是当KeepAlive agent没有被使用时，显式地 `destroy()` KeepAlive agent仍然是个好主意，这样 `socket` 会被关闭。

当 `socket` 触发了 `close` 事件或者特殊的 `agentRemove` 事件的时候，套接字们从agent的资源池中移除。这意味着如果你打算保持一个HTTP请求长时间开启，并且不希望它保持在资源池中，那么你可以按照下列几行的代码做事：

```
http.get(options, function(res) {
  // Do stuff
}).on("socket", function (socket) {
  socket.emit("agentRemove");
});
```

另外，你可以使用 `agent:false` 来停用池：

```
http.get({
  hostname: 'localhost',
  port: 80,
  path: '/',
  agent: false  // create a new agent just for this one request
}, function (res) {
  // Do stuff with response
})
```

new Agent([options])#

options Object 为**agent**设置可配置的选项。可以有以下属性:

- keepAlive Boolean 在未来保持池中的 `socket` 被其他请求所使用，默认为 `false`
- keepAliveMsecs Integer 当使用HTTP KeepAlive时，通过被保持连接的 `socket` 发送TCP KeepAlive 报文的间隔。默认为 `1000` 。只在 `KeepAlive` 被设置为 `true` 时有效
- maxSockets Number 每个主机允许拥有的 `socket` 的最大数量。默认为 `Infinity`

- maxFreeSockets Number 在空闲状态下允许打开的最大 `socket` 数。仅在 `keepAlive` 为 `true` 时有效。默认为 `256`

`http.request` 使用的默认的 `http.globalAgent` 包含它们属性的各自的默认值。

为了配置它们中的任何一个，你必须创建你自己的 `Agent` 对象。

```
var http = require('http');
var keepAliveAgent = new http.Agent({ keepAlive: true });
options.agent = keepAliveAgent;
http.request(options, onResponseCallback);
```

agent.maxSockets

默认为 `Infinity` 。决定了每个源上可以拥有的并发的 `socket` 的数量。源为 `'host:port'` 或 `'host:port:localAddress'` 结合体。

agent.maxFreeSockets

默认为 `256` 。对于支持HTTP KeepAlive的Agent，这设置了在空闲状态下保持打开的最大 `socket` 数量。

agent.sockets

这个对象包含了正在被Agent使用de `socket` 数组。请不要修改它。

agent.freeSockets

这个对象包含了当HTTP KeepAlive被使用时正在等待的 `socket` 数组。请不要修改它。

agent.requests

这个对象包含了还没有被分配给 `socket` 的请求队列。请不要修改它。

agent.destroy()

销毁正在被agent使用的所有 `socket` 。

通常没有必要这么做。但是，如果你正在使用一个启用了 `KeepAlive` 的agent，那么最好明确地关闭agent当你知道它不会再被使用时。否则，在服务器关闭它们前 `socket` 可能被闲置。

agent.getName(options)

通过一个请求选项集合来获取一个独一无二的名字，来决定一个连接是否可被再使用。在http代理中，这返回 `host:port:localAddress` 。在https代理中， `name` 包括了CA，cert，ciphers和 `HTTPS/TLS-specific` 配置来决定一个 `socket` 是否能被再使用。

http.globalAgent

所有的http客户端请求使用的默认全局 `Agent` 实例。

Class: http.ClientRequest

这个对象时被内部创建的，并且通过 `http.request()` 被返回。它代表了一个正在处理的请求，其头部已经进入了队列。这个头部仍然可以通过 `setHeader(name, value)` ， `getHeader(name)` 和 `removeHeader(name)` 修改。实际的头部会随着第一个数据块发送，或在关闭连接时发送。

要获得响应对象，请为 `response` 事件添加一个监听器。 `request` 对象的 `response` 事件将会在收到响应头时触发。这个 `response` 事件的第一个参数是一个 `http.IncomingMessage` 的实例。

在 `response` 事件期间，可以给响应对象添加监听器；尤其是监听 `data` 事件。

如果没有添加 `response` 事件监听器，那么响应会被完全忽略。但是，如果你添加了 `response` 事件，那么你必须通过调用 `response.read()` ，添加 `data` 事件监听器或调用 `.resume()` 方法等等，来从响应对象中消耗数据。在数据被消费之前， `end` 事件不会触发。如果数据没有被读取，它会消耗内存，最后导致 `'process out of memory'` 错误。

注意： `node.js` 不会检查 `Content-Length` 和被传输的响应体长度是否相同。

这个请求实现了 `Writable` 流接口。这是一个包含了以下事件的 `EventEmitter` ：

Event: 'response'

- function (response) { }

当这个请求收到一个响应时触发。这个事件只会被触发一次。 `response` 参数是一个 `http.IncomingMessage` 实例。

- **Options:**
  - host: 一个向其发送请求的服务器的域名或IP地址
  - port: 远程服务器的端口
  - socketPath: Unix域 `socket` （使用 `host:port` 或 `socketPath` 中的一个）

Event: 'socket'

- function (socket) { }

当一个 `socket` 被分配给一个请求时触发。

Event: 'connect'

- function (response, socket, head) { }

每次服务器使用 `CONNECT` 方法响应一个请求时触发。如果这个事件没有被监听，那么接受 `CONNECT` 方法的客户端将会关闭它们的连接。

以下是一对客户端/服务器代码，展示如何监听 `connect` 事件。

```
var http = require('http');
var net = require('net');
var url = require('url');

// Create an HTTP tunneling proxy
var proxy = http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('okay');
});
proxy.on('connect', function(req, cltSocket, head) {
  // connect to an origin server
  var srvUrl = url.parse('http://' + req.url);
  var srvSocket = net.connect(srvUrl.port, srvUrl.hostname, func
```

```
tion() {
    cltSocket.write('HTTP/1.1 200 Connection Established\r\n' +
                    'Proxy-agent: node.js-Proxy\r\n' +
                    '\r\n');
    srvSocket.write(head);
    srvSocket.pipe(cltSocket);
    cltSocket.pipe(srvSocket);
  });
});

// now that proxy is running
proxy.listen(1337, '127.0.0.1', function() {

  // make a request to a tunneling proxy
  var options = {
    port: 1337,
    hostname: '127.0.0.1',
    method: 'CONNECT',
    path: 'www.google.com:80'
  };

  var req = http.request(options);
  req.end();

  req.on('connect', function(res, socket, head) {
    console.log('got connected!');

    // make a request over an HTTP tunnel
    socket.write('GET / HTTP/1.1\r\n' +
                 'Host: www.google.com:80\r\n' +
                 'Connection: close\r\n' +
                 '\r\n');
    socket.on('data', function(chunk) {
      console.log(chunk.toString());
    });
    socket.on('end', function() {
      proxy.close();
    });
  });
});
```

Event: 'upgrade'

- function (response, socket, head) { }

Emitted each time a server responds to a request with an upgrade. If this event isn't being listened for, clients receiving an upgrade header will have their connections closed. 每次服务器返回 `upgrade` 响应给请求时触发。如果这个事件没有被监听，客户端接收一个 `upgrade` 头时会关闭它们的连接。

以下是一对客户端/服务器代码，展示如何监听 `upgrade` 事件。

```
var http = require('http');

// Create an HTTP server
var srv = http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('okay');
});
srv.on('upgrade', function(req, socket, head) {
  socket.write('HTTP/1.1 101 Web Socket Protocol Handshake\r\n'
+
               'Upgrade: WebSocket\r\n' +
               'Connection: Upgrade\r\n' +
               '\r\n');

  socket.pipe(socket); // echo back
});

// now that server is running
srv.listen(1337, '127.0.0.1', function() {

  // make a request
  var options = {
    port: 1337,
    hostname: '127.0.0.1',
    headers: {
      'Connection': 'Upgrade',
      'Upgrade': 'websocket'
    }
```

```
  };

  var req = http.request(options);
  req.end();

  req.on('upgrade', function(res, socket, upgradeHead) {
    console.log('got upgraded!');
    socket.end();
    process.exit(0);
  });
});
```

Event: 'continue'

- function () { }

当服务器发出一个 `'100 Continue'` HTTP响应时，通常这是因为请求包含 `'Expect: 100-continue'` 。这是一个客户端须要发送请求体的指示。

Event: 'abort'

- function () { }

当请求被客户端中止时触发。这个事件只会在第一次调用 `abort()` 时触发。

request.flushHeaders()

冲刷请求头。

由于效率原因， `node.js` 通常在直到你调用 `request.end()` 或写入第一个数据块前都会缓冲请求头，然后努力将请求头和数据打包为一个TCP报文。

这通常是你想要的（它节约了一个TCP往返）。但当第一份数据会等待很久才被发送时不是。 `request.flushHeaders()` 使你能绕过这个优化并且启动请求。

request.write(chunk[, encoding][, callback])

发送一个响应块。当用户想要将请求体流式得发送给服务器时，可以通过调用这个方法多次来办到--在这种情况下，建议在创建请求时使用 `['Transfer-Encoding', 'chunked']` 头。

`chunk` 参数必须是一个 `Buffer` 或一个字符串。

`encoding` 参数是可选的，并且仅当 `chunk` 是字符串时有效。默认为 `'utf8'` 。

`callback` 参数是可选的，并且当数据块被冲刷时被调用。

request.end([data][, encoding][, callback])

结束发送请求。如果有任何部分的请求体未被发送，这个函数将会将它们冲刷至流中。如果请求是成块的，它会发送终结符 `'0\r\n\r\n'` 。

如果 `data` 被指定，那么这与调用 `request.write(data, encoding)` 后再调用 `request.end(callback)` 相同。

如果 `callback` 被指定，那么它将在请求流结束时被调用。

request.abort()

中止请求。

request.setTimeout(timeout[, callback])

一旦一个 `socket` 被分配给这个请求并且完成连接， `socket.setTimeout()` 会被调用。

返回 `request` 对象。

request.setNoDelay([noDelay])

一旦一个 `socket` 被分配给这个请求并且完成连接， `socket.setNoDelay()` 会被调用。

request.setSocketKeepAlive([enable][, initialDelay])

一旦一个 `socket` 被分配给这个请求并且完成连接， `socket.setKeepAlive()` 会被调用。

http.IncomingMessage

一个 `IncomingMessage` 对象被 `http.Server` 或 `http.ClientRequest` 创建，并且分别被传递给 `request` 和 `response` 事件的第一个参数。它被用来取得响应状态，响应头和响应体。

它实现了 `Readable` 流接口，并且有以下额外的事件，方法和属性。

Event: 'close'

- function () { }

表明底层连接被关闭。与 `end` 相同，这个时间每次响应只会触发一次。

message.httpVersion

当向服务器发送请求时，客户端发送的HTTP版本。向客户端发送响应时，服务器响应的HTTP版本。通常是 `'1.1'` 或 `'1.0'` 。

另外， `response.httpVersionMajor` 是第一个整数， `response.httpVersionMinor` 是第二个整数。

message.headers

请求/响应头对象。

只读的头名称和值映射。头名称是小写的，例子：

```
// Prints something like:
//
// { 'user-agent': 'curl/7.22.0',
//   host: '127.0.0.1:8000',
//   accept: '*/*' }
console.log(request.headers);
```

message.rawHeaders

接受到的原始请求/响应头列表。

注意键和值在同一个列表中，它并非一个元组列表。于是，偶数偏移量为键，奇数偏移量为对应的值。

头名称不是必须小写的，并且重复也没有被合并。

```
// Prints something like:
//
// [ 'user-agent',
//   'this is invalid because there can be only one',
//   'User-Agent',
//   'curl/7.22.0',
//   'Host',
//   '127.0.0.1:8000',
//   'ACCEPT',
//   '*/*' ]
console.log(request.rawHeaders);
```

message.trailers

请求/响应尾部对象。只在 `end` 事件中存在。

message.rawTrailers

接受到的原始请求/响应头尾部键值对。只在 `end` 事件中存在。

message.setTimeout(msecs, callback)

- msecs Number
- callback Function

调用 `message.connection.setTimeout(msecs, callback)`。

返回 `message`。

message.method

仅对从 `http.Server` 获得的请求有效。

请求方法是字符串。只读。例如：`'GET'`，`'DELETE'`。

message.url

仅对从 `http.Server` 获得的请求有效。

请求的URL字符串。这仅仅只包含实际HTTP请求中的URL。如果请求是：

```
GET /status?name=ryan HTTP/1.1\r\n
Accept: text/plain\r\n
\r\n
```

那么 `request.url` 将是：

```
'/status?name=ryan'
```

如果你想分块地解释URL。你可以调用 `require('url').parse(request.url)` 。例子：

```
iojs> require('url').parse('/status?name=ryan')
{ href: '/status?name=ryan',
  search: '?name=ryan',
  query: 'name=ryan',
  pathname: '/status' }
```

如果你想从查询字符串中提取参数，你可以使用 `require('querystring').parse` 函数，或者给 `require('url').parse` 方法的第二个参数传递 `true` ，例子：

```
iojs> require('url').parse('/status?name=ryan', true)
{ href: '/status?name=ryan',
  search: '?name=ryan',
  query: { name: 'ryan' },
  pathname: '/status' }
```

message.statusCode

只对从 `http.ClientRequest` 到来的响应有效。

3位整数HTTP状态码。如 `404` 。

message.statusMessage

只对从 `http.ClientRequest` 到来的响应有效。

HTTP响应状态信息。如 `OK` 或 `Internal Server Error` 。

message.socket

与此连接关联的 `net.Socket` 对象。

通过HTTPS的支持，使用 `request.socket.getPeerCertificate()` 来获取客户端的身份细节。

HTTPS

稳定度: 2 - 稳定

HTTPS是建立在TLS/SSL之上的HTTP协议。在 `node.js` 中，它被作为单独模块实现。

Class: https.Server

这个类是 `tls.Server` 的子类，并且和 `http.Server` 触发相同的事件。更多信息请参阅 `http.Server` 。

server.setTimeout(msecs, callback)

参阅 `http.Server#setTimeout()` 。

server.timeout

参阅 `http.Server#timeout` 。

https.createServer(options[, requestListener])

返回一个新的HTTPS web服务器对象。 `options` 与 `tls.createServer()` 中的类似。 `requestListener` 会被自动添加为 `request` 事件的监听器。

例子：

```
// curl -k https://localhost:8000/
var https = require('https');
var fs = require('fs');

var options = {
  key: fs.readFileSync('test/fixtures/keys/agent2-key.pem'),
  cert: fs.readFileSync('test/fixtures/keys/agent2-cert.pem')
};

https.createServer(options, function (req, res) {
  res.writeHead(200);
  res.end("hello world\n");
}).listen(8000);
```

或

```
var https = require('https');
var fs = require('fs');

var options = {
  pfx: fs.readFileSync('server.pfx')
};

https.createServer(options, function (req, res) {
  res.writeHead(200);
  res.end("hello world\n");
}).listen(8000);
```

server.listen(port[, host][, backlog][, callback])

server.listen(path[, callback])

server.listen(handle[, callback])

详情参阅 `http.listen()` 。

server.close([callback])

详情参阅 `http.close()` 。

https.request(options, callback)

向一个安全web服务器发送请求。

`options` 可以是一个对象或一个字符串。如果 `options` 是一个字符串，它会自动被 `url.parse()` 解析。

所有的 `http.request()` 选项都是可用的。

例子：

```
var https = require('https');

var options = {
  hostname: 'encrypted.google.com',
  port: 443,
  path: '/',
  method: 'GET'
};

var req = https.request(options, function(res) {
  console.log("statusCode: ", res.statusCode);
  console.log("headers: ", res.headers);

  res.on('data', function(d) {
    process.stdout.write(d);
  });
});
req.end();

req.on('error', function(e) {
  console.error(e);
});
```

`options` 参数有以下选项：

- host: 一个将要向其发送请求的服务器域名或IP地址。默认为 `localhost` 。
- hostname: `host` 的别名。为了支持 `url.parse()` 的

话，`hostname` 比 `host` 更好些。
- family: 解析 `host` 和 `hostname` 时的IP地址协议族。合法值是 `4` 和 `6` 。当没有指定时，将都被使用。
- port: 远程服务器端口。默认为 `80` 。
- localAddress: 用于绑定网络连接的本地端口。
- socketPath: Unix域 `socket` （使用 `host:port` 或 `socketPath` ）。
- method: 指定HTTP请求方法的字符串。默认为 `GET` 。
- path: 请求路径。默认为 `/` 。如果有查询字符串，则需要包含。例如'/index.html?page=12'。请求路径包含非法字符时抛出异常。目前，只否决空格，不过在未来可能改变。
- headers: 一个包含请求头的对象。
- auth: 用于计算认证头的基本认证，即 `'user:password'` 。
- **agent**: 控制 `agent` 行为。当使用一个代理时，请求将默认为 `Connection: keep-alive` 。可能值有：
  - undefined (默认): 在这个主机和端口上使用全局`agent。
  - Agent object: 在 `agent` 中显示使用passed。
  - false: 跳出 `agent` 的连接池。默认请求为 `Connection: close` 。

以下来自 `tls.connect()` 的选项也可以被指定。但是，一个 `globalAgent` 会默默忽略这些。

- pfx: 证书，SSL所用的私钥和CA证书。默认为 `null` 。
- key: SSL所用的私钥。默认为 `null` 。
- passphrase: 私钥或pfx的口令字符串。默认为 `null` 。
- cert: 所用的公共x509证书。默认为 `null` 。
- ca: 一个用来检查远程主机的权威证书或权威证书数组。
- ciphers: 一个描述要使用或排除的密码的字符串。更多格式信息请查询 `http://www.openssl.org/docs/apps/ciphers.html#CIPHER_LIST_FORMAT` 。
- rejectUnauthorized: 如果设置为 `true` ，服务器证书会使用所给的CA列表验证。验证失败时，一个 `error` 事件会被触发。验证发生于连接层，在HTTP请求发送之前。默认为 `true` 。
- secureProtocol: 所用的SSL方法，如 `SSLv3_method` 强制使用SSL v3。可用的值取决你的OpenSSL安装和 `SSL_METHODS` 常量。

要指定这些选项，使用一个自定义的 `Agent` 。

例子：

```
var options = {
  hostname: 'encrypted.google.com',
  port: 443,
  path: '/',
  method: 'GET',
  key: fs.readFileSync('test/fixtures/keys/agent2-key.pem'),
  cert: fs.readFileSync('test/fixtures/keys/agent2-cert.pem')
};
options.agent = new https.Agent(options);

var req = https.request(options, function(res) {
  ...
}
```

或不使用 `Agent` 。

例子：

```
var options = {
  hostname: 'encrypted.google.com',
  port: 443,
  path: '/',
  method: 'GET',
  key: fs.readFileSync('test/fixtures/keys/agent2-key.pem'),
  cert: fs.readFileSync('test/fixtures/keys/agent2-cert.pem'),
  agent: false
};

var req = https.request(options, function(res) {
  ...
}
```

https.get(options, callback)

类似于 `http.get()` ，但是使用HTTPS。

`options` 可以是一个对象或一个字符串。如果 `options` 是一个字符串，它会自动被 `url.parse()` 解析。

例子：

```
var https = require('https');

https.get('https://encrypted.google.com/', function(res) {
  console.log("statusCode: ", res.statusCode);
  console.log("headers: ", res.headers);

  res.on('data', function(d) {
    process.stdout.write(d);
  });

}).on('error', function(e) {
  console.error(e);
});
```

Class: https.Agent

一个与 `http.Agent` 类似的HTTPS `Agent` 对象。更多信息请参阅 `https.request()` 。

https.globalAgent

所有HTTPS客户端请求的全局 `https.Agent` 实例。

Modules

稳定度: 3 - 锁定

`node.js` 又一个简单的模块加载系统。在 `node.js` 中，文件和模块是一一对应的。以下例子中， `foo.js` 加载的同目录下的 `circle.js` 。

`foo.js` 的内容：

```
var circle = require('./circle.js');
console.log( 'The area of a circle of radius 4 is '
           + circle.area(4));
```

`circle.js` 的内容：

```
var PI = Math.PI;

exports.area = function (r) {
  return PI * r * r;
};

exports.circumference = function (r) {
  return 2 * PI * r;
};
```

`circle.js` 模块暴露了 `area()` 函数和 `circumference()` 函数。想要为你的模块添加函数或对象，你可以将它们添加至特殊的 `exports` 对象的属性上。

模块的本地变量是私有的，好似模块被包裹在一个函数中。在这个例子中变量 `PI` 是 `circle.js` 私有的。

如果想要你的模块暴露一个函数（例如一个构造函数），或者想要一次赋值就暴露一个完整的对象，而不是一次绑定一个属性，那就将之赋值给 `module.exports` 而不是 `exports` 。

以下， `bar.js` 使用了暴露了一个构造函数的 `square` 模块：

```
var square = require('./square.js');
var mySquare = square(2);
console.log('The area of my square is ' + mySquare.area());
```

`square` 模块内部：

```
// assigning to exports will not modify module, must use module.
exports
module.exports = function(width) {
  return {
    area: function() {
      return width * width;
    }
  };
}
```

模块系统在 `require("module")` 中被实现。

循环依赖

当存在循环的 `require()` 调用。一个模块可能在返回时，被没有被执行完毕。

考虑一下情况：

`a.js` ：

```
console.log('a starting');
exports.done = false;
var b = require('./b.js');
console.log('in a, b.done = %j', b.done);
exports.done = true;
console.log('a done');
```

`b.js` ：

```
console.log('b starting');
exports.done = false;
var a = require('./a.js');
console.log('in b, a.done = %j', a.done);
exports.done = true;
console.log('b done');
```

`main.js` ：

```
console.log('main starting');
var a = require('./a.js');
var b = require('./b.js');
console.log('in main, a.done=%j, b.done=%j', a.done, b.done);
```

当 `main.js` 加载 `a.js` ，而后 `a.js` 会去加载 `b.js` 。与此同时， `b.js` 尝试去加载 `a.js` 。为了避免一个无限循环， `a.js` 会返回一个未完成的副本给 `b.js` 模块。 `b.js` 会接着完成加载，然后它所暴露的值再被提供给 `a.js` 模块。

这样 `main.js` 就完成了它们的加载。因此程序的输出是：

```
$ iojs main.js
main starting
a starting
b starting
in b, a.done = false
b done
in a, b.done = true
a done
in main, a.done=true, b.done=true
```

如果在你的程序里有循环依赖，请确保它们按你的计划工作。

核心模块

`node.js` 中有一些模块是被编译成二进制的。这些模块会在本文档的其他地方详细讨论。

核心模块被定义在 `node.js` 源码的 `lib/` 目录下。

当被 `require()` 时，核心模块总是被优先加载的。例如 `require('http')` 总是会返回内建的HTTP模块，甚至是有一个同名文件时。

文件模块

如果准确的文件名没有被发现，那么 `node.js` 将会依次添加 `.js` ， `.json` 或 `.node` 后缀名，然后试图去加载。

`.js` 文件被解释为 `JavaScript` 文本文件， `.json` 被解释为 `JSON` 文本文件， `.node` 文件被解释为编译好的插件模块，然后被 `dlopen` 加载。

前缀是 `'/'` 则是文件的绝对路径。例如 `require('/home/marco/foo.js')` 将会加载 `/home/marco/foo.js` 。

前缀是 `'./'` 则是调用 `require()` 的文件的相对路径。也就是说， `circle.js` 必须与 `foo.js` 在同一目录下，这样 `require('./circle')` 才能找到它。

如果没有 `'/'` ， `'./'` 或 `'../'` 前缀，模块要么是一个核心模块，或是需要从 `node_modules` 目录中被加载。

如果指定的路径不存在， `require()` 将会抛出一个 `code` 属性是 `'MODULE_NOT_FOUND'` 的错误。

从node_modules目录中加载

如果传递给 `require()` 的模块标识符不是一个本地模块，也没有以 `'/'` ， `'../'` 或 `'./'` 开始。那么 `node.js` 将会从当前目录的父目录开始，添加 `/node_modules` ，试图从这个路径来加载模块。

如果还是没有找到模块，那么它会再移至此目录的父目录，如此往复，直至到达文件系统的根目录。

例如，如果一个位于 `'/home/ry/projects/foo.js'` 的文件调用了 `require('bar.js')` ，那么 `node.js` 将会按照以下的路径顺序来查找：

```
/home/ry/projects/node_modules/bar.js
/home/ry/node_modules/bar.js
/home/node_modules/bar.js
/node_modules/bar.js
```

这要求程序本地化（localize）自己的依赖，防止它们崩溃。

你也可以在模块名中加入一个路径后缀，来引用这个模块中特定的一个文件或子模块。例如， `require('example-module/path/to/file')` 将会从 `example-module` 的位置解析相对路径 `path/to/file` 。路径后缀遵循相同的模块解析语义。

作为模块的目录

在一个单独目录下组织程序和库，然后提供一个单独的入口，是非常便捷的。有三种方法，可以将目录作为 `require()` 的参数，来加载模块。

第一种方法是，在模块的根目录下创建一个 `package.json` 文件，其中指定了 `main` 模块。一个示例 `package.json` 文件：

```
{ "name" : "some-library",
  "main" : "./lib/some-library.js" }
```

如果这个文件位于 `./some-library` ，那么 `require('./some-library')` 将会试图去加载 `./some-library/lib/some-library.js` 。

这就是 `node.js` 所能够了解 `package.json` 文件的程度。

如果目录中没有 `package.json` 文件，那么 `node.js` 将会视图去加载当前目录中的 `index.js` 或 `index.node` 。例如，如果在上面的例子中没有 `package.json` ，那么 `require('./some-library')` 将会试图加载：

```
./some-library/index.js
./some-library/index.node
```

缓存

模块在第一次被加载后，会被缓存。这意味着，如果都解析到了相同的文件，每一次调用 `require('foo')` 都将会返回同一个对象。

多次调用 `require('foo')` 可能不会造成模块代码被执行多次。这是一个重要的特性。有了它，“部分完成”的对象也可以被返回，这样，传递依赖也能被加载，即使它们可能会造成循环依赖。

如果你想要一个模块被多次执行，那么就暴露一个函数，然后执行这个函数。

模块缓存警告

模块的缓存依赖于它们被解析后的文件名。所以调用模块的位置不同，可以会解析出不同的文件名（比如需要从node_modules目录中加载）。所以不能保证 `require('foo')` 总是会返回相同的对象，因为它们可能被解析为了不同的文件。

module对象

- {Object}

每一个模块中，变量 `module` 是一个代表了当前模块的引用。为了方便， `module.exports` 也可以通过模块作用域中的 `exports` 取得。 `module` 对象实际上不是全局的，而是每个模块本地的。

module.exports

- Object

`module.exports` 对象是由模块系统创建的。有时这是难以接受的；许多人希望它们的模块是一些类的实例。如果需要这样，那么就将想要暴露的对象赋值给 `module.exports` 。注意，将想要暴露的对象传递给 `exports` ，将仅仅只会重新绑定（rebind）本地变量 `exports` ，所以不要这么做。

例如假设我们正在写一个叫做 `a.js` 的模块：

```
var EventEmitter = require('events').EventEmitter;

module.exports = new EventEmitter();

// Do some work, and after some time emit
// the 'ready' event from the module itself.
setTimeout(function() {
  module.exports.emit('ready');
}, 1000);
```

那么在另一个文件中我们可以：

```
var a = require('./a');
a.on('ready', function() {
  console.log('module a is ready');
});
```

主要，对 `module.exports` 的赋值必须立刻完成。它不能在任何的回调函数中完成。以下例子将不能正常工作：

`x.js` ：

```
setTimeout(function() {
  module.exports = { a: "hello" };
}, 0);
```

`y.js` ：

```
var x = require('./x');
console.log(x.a);
```

exports快捷方式

`exports` 变量是一个 `module.exports` 的引用。如果你将一个新的值赋予它，那么它将不再指向先前的那个值。

为了说明这个行为，将 `require()` 的实现假设为这样：

```
function require(...) {
  // ...
  function (module, exports) {
    // Your module code here
    exports = some_func;        // re-assigns exports, exports i
s no longer
                                // a shortcut, and nothing is ex
ported.
    module.exports = some_func; // makes your module export 0
  } (module, module.exports);
  return module;
}
```

一个指导方针是，如果你弄不清楚 `exports` 和 `module.exports` 之间的关系，请只使用 `module.exports` 。

module.require(id)

- id String
- Return: 被解析的模块的 `module.exports`

`module.require` 方法提供了一种像 `require()` 一样，从源模块中加载模块的方法。

注意，为了这么做，你必须取得 `module` 对象的引用。因为 `require()` 返回 `module.exports` ，并且 `module` 对象是一个典型的只在特定的模块作用域中有效的变量，如果要使用它，必须被明确地导出。

module.id

- String

模块的识别符。通常是被完全解析的文件名。

module.filename

- String

模块完全解析后的文件名。

module.loaded

- Boolean

模块是否加载完成，或者是正在加载的过程中。

module.parent

- Module Object

引用这个模块的模块。

module.children

- Array

这个模块所引入的模块。

总体来说

为了获得 `require()` 被调用时将要被加载的准确文件名，使用 `require.resolve()` 函数。

综上所述，以下是一个 `require.resolve` 所做的事的高级算法伪代码：

```
require(X) from module at path Y
1. If X is a core module,
   a. return the core module
   b. STOP
2. If X begins with './' or '/' or '../'
   a. LOAD_AS_FILE(Y + X)
   b. LOAD_AS_DIRECTORY(Y + X)
3. LOAD_NODE_MODULES(X, dirname(Y))
4. THROW "not found"

LOAD_AS_FILE(X)
1. If X is a file, load X as JavaScript text.  STOP
2. If X.js is a file, load X.js as JavaScript text.  STOP
3. If X.json is a file, parse X.json to a JavaScript Object.  ST
OP
4. If X.node is a file, load X.node as binary addon.  STOP
```

```
LOAD_AS_DIRECTORY(X)
1. If X/package.json is a file,
   a. Parse X/package.json, and look for "main" field.
   b. let M = X + (json main field)
   c. LOAD_AS_FILE(M)
2. If X/index.js is a file, load X/index.js as JavaScript text.
 STOP
3. If X/index.json is a file, parse X/index.json to a JavaScript
 object. STOP
4. If X/index.node is a file, load X/index.node as binary addon.
  STOP

LOAD_NODE_MODULES(X, START)
1. let DIRS=NODE_MODULES_PATHS(START)
2. for each DIR in DIRS:
   a. LOAD_AS_FILE(DIR/X)
   b. LOAD_AS_DIRECTORY(DIR/X)

NODE_MODULES_PATHS(START)
1. let PARTS = path split(START)
2. let I = count of PARTS - 1
3. let DIRS = []
4. while I >= 0,
   a. if PARTS[I] = "node_modules" CONTINUE
   c. DIR = path join(PARTS[0 .. I] + "node_modules")
   b. DIRS = DIRS + DIR
   c. let I = I - 1
5. return DIRS
```

从全局文件夹加载

如果 `NODE_PATH` 环境变量被设置为了一个以冒号分割的绝对路径列表，那么在找不到模块时， `node.js` 将会从这些路径中寻找模块（注意：在Windows中， `NODE_PATH` 是以分号间隔的）。

`NODE_PATH` 最初被创建，是用来支持在当前的模块解析算法被冻结（frozen）前，从不同的路径加载模块的。

`NODE_PATH` 仍然被支持，但是，如今 `node.js` 生态圈已经有了放置依赖模块的公约，它已经不那么必要的。有时，当人们没有意识到 `NODE_PATH` 有被设置时，依赖于 `NODE_PATH` 的部署可能会产生出人意料的表现。有时，一个模块的依赖改变了，造成了通过 `NODE_PATH` ，加载了不同版本的模块。

另外， `node.js` 将会查找以下路径：

- 1: $HOME/.node_modules
- 2: $HOME/.node_libraries
- 3: $PREFIX/lib/node

`$HOME` 是用户的家目录， `$PREFIX` 是 `node.js` 中配置的 `node_prefix` 。

由于一些历史原因，高度推荐你将依赖放入 `node_modules` 目录。它会被加载的更快，且可靠性更好。

访问主模块

当一个文件直接由 `node.js` 执行， `require.main` 将被设置为这个模块。这意味着你可以判断一个文件是否是直接被运行的。

```
require.main === module
```

对于一个文件 `foo.js` ，如果通过 `iojs foo.js` 运行，以上将会返回 `true` 。如果通过 `require('./foo')` ，将会返回 `false` 。

因为 `module` 提供了一个 `filename` 属性（通常等于 `__filename` ），所以当前应用的入口点可以通过检查 `require.main.filename` 来获取。

附录：包管理小贴士

`node.js` 的 `require()` 函数的语义被设计得足够通用，来支持各种目录结构。包管理程序诸如 `dpkg` ， `rpm` 和 `npm` 将可以通过不修改 `node.js` 模块，来构建本地包。

以下我们给出一个建议的可行的目录结构：

假设 `/usr/lib/node/<some-package>/<some-version>` 中有指定版本包的内容。

包可以依赖于其他包。为了安装 `foo` 包，你可能需要安装特定版本的 `bar` 包。 `bar` 包可能有它自己的依赖，在一些情况下，它们的依赖可以会冲突或者产生循环。

由于 `node.js` 会查找任何它加载的包得眞实路径（也就是说，解析 `symlinks` ），解析以下结构的方案非常简单：

- /usr/lib/node/foo/1.2.3/ - `foo` 包的内容， `1.2.3` 版本。
- /usr/lib/node/bar/4.3.2/ - `foo` 包所依赖的 `bar` 包的内容。
- /usr/lib/node/foo/1.2.3/node_modules/bar - 指向 `/usr/lib/node/bar/4.3.2/` 的符号链接。
- /usr/lib/node/bar/4.3.2/node_modules/* - 指向 `bar` 包所依赖的包的符号链接。

因此，即使有循环依赖，或者依赖冲突，每个模块都能够获取它们使用的特定版本的依赖。

当 `foo` 包中的代码执行 `require('bar')` ，将会获得符号链接 `/usr/lib/node/foo/1.2.3/node_modules/bar` 指向的版本。接着， `bar` 包种的代码执行 `require('quux')` ，它将会获得符号链接 `/usr/lib/node/bar/4.3.2/node_modules/quux` 指向的版本。

此外，为了优化模块查找的过程，我们将模块放在 `/usr/lib/node_modules/<name>/<version>` 而不是直接放在 `/usr/lib/node` 中。然后在找不到依赖时， `node.js` 就不会一直去查找 `/usr/node_modules` 或 `/node_modules` 目录了。

为了让模块在 `node.js` 的REPL中可用，可能需要将 `/usr/lib/node_modules` 目录加入到 `$NODE_PATH` 环境变量。因为使用 `node_modules` 目录的模块查找都是使用相对路径，且基于调用 `require()` 的文件的眞实路径，因此包本身可以在任何位置。

net

稳定度: 2 - 稳定

`net` 模块为你提供了异步的网络调用的包装。它同时包含了创建服务器和客户端的函数。你可以通过 `require('net')` 来引入这个模块。

net.createServer([options][, connectionListener])

创建一个新的TCP服务器。 `connectionListener` 参数会被自动绑定为 `connection` 事件的监听器。

`options` 是一个包含下列默认值的对象：

```
{
  allowHalfOpen: false,
  pauseOnConnect: false
}
```

如果 `allowHalfOpen` 是 `true` ，那么当另一端的 `socket` 发送一个 `FIN` 报文时 `socket` 并不会自动发送 `FIN` 报文。 `socket` 变得不可读，但是可写。你需要明确地调用 `end()` 方法。详见 `end` 事件。

如果 `pauseOnConnect` 是true，那么 `socket` 在每一次被连接时会暂停，并且不会读取数据。这允许在进程间被传递的连接不读取任何数据。如果要让一个被暂停的 `socket` 开始读取数据，调用 `resume()` 方法。

以下是一个应答服务器的例子，监听8124端口：

```
var net = require('net');
var server = net.createServer(function(c) { //'connection' liste
ner
  console.log('client connected');
  c.on('end', function() {
    console.log('client disconnected');
  });
  c.write('hello\r\n');
  c.pipe(c);
});
server.listen(8124, function() { //'listening' listener
  console.log('server bound');
});
```

使用 `telnet` 测试：

```
telnet localhost 8124
```

想要监听 `socket``/tmp/echo.sock` ，只需改变倒数第三行：

```
server.listen('/tmp/echo.sock', function() { //'listening' liste
ner
```

使用 `nc` 连接一个UNIX domain socket服务器：

```
nc -U /tmp/echo.sock
```

net.connect(options[, connectionListener])

net.createConnection(options[, connectionListener])

工厂函数，返回一个新的 `net.Socket` 实例，并且自动使用提供的 `options` 进行连接。

`options` 会被同时传递给 `net.Socket` 构造函数和 `socket.connect` 方法。

参数 `connectListener` 将会被立即添加为 `connect` 事件的监听器。

下面是一个上文应答服务器的客户端的例子：

```
var net = require('net');
var client = net.connect({port: 8124},
    function() { //'connect' listener
  console.log('connected to server!');
  client.write('world!\r\n');
});
client.on('data', function(data) {
  console.log(data.toString());
  client.end();
});
client.on('end', function() {
  console.log('disconnected from server');
});
```

要连接 `socket``/tmp/echo.sock` 只需要改变第二行为：

```
var client = net.connect({path: '/tmp/echo.sock'});
```

net.connect(port[, host][, connectListener])

net.createConnection(port[, host][, connectListener])

工厂函数，返回一个新的 `net.Socket` 实例，并且自动使用指定的端口(port)和主机(host)进行连接。

如果 `host` 被省略，默认为 `localhost` 。

参数 `connectListener` 将会被立即添加为 `connect` 事件的监听器。

net.connect(path[, connectListener])

net.createConnection(path[, connectListener])

工厂函数，返回一个新的unix `net.Socket` 实例，并且自动使用提供的路径(path)进行连接。

参数 `connectListener` 将会被立即添加为 `connect` 事件的监听器。

Class: net.Server

这个类用于创建一个TCP或本地服务器。

server.listen(port[, hostname][, backlog][, callback])

开始从指定端口和主机名接收连接。如果省略主机名，那么如果IPv6可用，服务器会接受从任何IPv6地址（::）来的链接，否则为任何IPv4地址（0.0.0.0）。如果端口为0那么将会为其设置一个随机端口。

积压量 `backlog` 是连接等待队列的最大长度。实际长度由你的操作系统的 `sysctl` 设置决定（如linux中的 `tcp_max_syn_backlog` 和 `somaxconn` ）。这个参数的默认值是511（不是512）。

这个函数式异步的。当服务器绑定了指定端口后， `listening` 事件将会被触发。最后一个参数 `callback` 将会被添加为 `listening` 事件的监听器。

有些用户可能遇到的情况是收到 `EADDRINUSE` 错误。这意味着另一个服务器已经使用了该端口。一个解决的办法是等待一段时间后重试。

```
server.on('error', function (e) {
  if (e.code == 'EADDRINUSE') {
    console.log('Address in use, retrying...');
    setTimeout(function () {
      server.close();
      server.listen(PORT, HOST);
    }, 1000);
  }
});
```

（注意， `node.js` 中所有的 `socket` 都已经设置了 `SO_REUSEADDR` ）

server.listen(path[, callback])

- path String
- callback Function

启动一个本地 `socket` 服务器，监听指定路径（ `path` ）上的连接。

这个函数式异步的。当服务器监听了指定路径后， `listening` 事件将会被触发。最后一个参数 `callback` 将会被添加为 `listening` 事件的监听器。

在UNIX中， `local domain` 经常被称作 `UNIX domain` 。 `path` 是一个文件系统路径名。它在被创建时会受相同文件名约定(same naming conventions)的限制并且进行权限检查(permissions checks)。它在文件系统中可见，并且在被删除前持续存在。

在Windows中， `local doamin` 使用一个命名管道（named pipe）实现。 `path` 必须指向 `\\?\pipe\` 或 `\\.\pipe\.` 中的一个条目，但是后者可能会做一些命名管道的处理，如处理 `..` 序列。除去表现，命名管道空间是平坦的（flat）。管道不会持续存在，它们将在最后一个它们的引用关闭后被删除。不要忘记，由于 `JavaScript` 的字符串转义，你必须在指定 `path` 时使用双反斜杠：

```
net.createServer().listen(
    path.join('\\\\?\\pipe', process.cwd(), 'myctl'))
```

server.listen(handle[, callback])

- handle Object
- callback Function

`handle` 对象可以被设置为一个服务器或一个 `socket` （或者任意以下划线开头的成员 `_handle` ），或者一个 `{fd: <n>}` 对象。

这将使得服务器使用指定句柄接受连接，但它假设文件描述符或句柄已经被绑定至指定的端口或域名 `socket` 。

在Windows下不支持监听一个文件描述符。

这个函数式异步的。当服务器已被绑定后， `listening` 事件将会被触发。最后一个参数 `callback` 将会被添加为 `listening` 事件的监听器。

server.listen(options[, callback])

- **options Object**
  - port Number 可选
  - host String 可选
  - backlog Number 可选

    - path String 可选
    - exclusive Boolean 可选
- callback Function 可选

`port` ，`host` 和 `backlog` 属性，以及可选的 `callback` 函数，与 `server.listen(port, [host], [backlog], [callback])` 中表现一致。 `path` 可以被指定为一个UNIX `socket` 。

如果 `exclusive` 是 `false` （默认），那么工作集群（cluster workers）将会使用相同的底层句柄，处理的连接的职责将会被它们共享。如果 `exclusive` 是 `true` ，那么句柄是不被共享的，企图共享将得到一个报错的结果。下面是一个监听独有端口的例子：

```
server.listen({
  host: 'localhost',
  port: 80,
  exclusive: true
});
```

server.close([callback])

使服务器停止接收新的连接并且保持已存在的连接。这个函数式异步的，当所有的连接都结束时服务器会最终关闭，并处罚一个 `close` 事件。可选的，你可以传递一个回调函数来监听 `close` 事件。如果传递了，那么它的唯一的第一个参数将表示任何可能潜在发生的错误。

server.address()

返回服务器绑定的地址，协议族名和端口通过操作系统报告。对查找操作系统分配的地址哪个端口被分配非常有用。返回一个有三个属性的对象。如 `{ port: 12346, family: 'IPv4', address: '127.0.0.1' }` 。

例子：

```
var server = net.createServer(function (socket) {
  socket.end("goodbye\n");
});

// grab a random port.
server.listen(function() {
  address = server.address();
  console.log("opened server on %j", address);
});
```

在 `listening` 事件触发前，不要调用 `server.address()` 方法。

server.unref()

调用一个 `server` 对象的 `unref` 方法将允许如果它是事件系统中唯一活跃的服务器，程序将会退出。如果服务器已经被调用过这个方法，那么再次调用这个方法将不会有任何效果。

返回 `server` 对象。

server.ref()

与 `unref` 相反，在一个已经被调用 `unref` 方法的 `server` 中调用 `ref` 方法，那么如果它是唯一活跃的服务器时，程序将不会退出（默认）。如果服务器已经被调用过这个方法，那么再次调用这个方法将不会有任何效果。

返回 `server` 对象。

server.maxConnections

设置了这个属性后，服务器的连接数达到时将会开始拒绝连接。

一旦 `socket` 被使用 `child_process.fork()` 传递给了子进程，这个属性就不被推荐去设置。

server.connections

这个函数已经被弃用。请使用 `server.getConnections()` 替代。

服务器的当前连接数。

当使用 `child_process.fork()` 传递一个 `socket` 给子进程时，这个属性将变成 `null` 。想要得到正确的结果请使用 `server.getConnections` 。

server.getConnections(callback)

异步地去获取服务器的当前连接数，在 `socket` 被传递给子进程时仍然可用。

回调函数的两个参数是 `err` 和 `count` 。

`net.Server` 是一个具有以下事件的 `EventEmitter` ：

Event: 'listening'

当调用 `server.listen` 后，服务器已被绑定时触发。

Event: 'connection'

- Socket object 连接对象

当新的连接产生时触发。 `socket` 是一个 `net.Socket` 实例。

Event: 'close'

当服务器关闭时触发。注意如果服务器中仍有连接存在，那么这个事件会直到所有的连接都关闭后才触发。

Event: 'error'

- Error Object

当发生错误时触发。 `close` 事件将会在它之后立即触发。参阅 `server.listen` 。

Class: net.Socket

这个对象是一个TCP或本地 `socket` 的抽象。 `net.Socket` 实例实现了双工流（duplex Stream）接口。它可以被使用者创建，并且被作为客户端（配合 `connect()` ）使用。或者也可以被 `node.js` 创建，并且通过服务器的 `connection` 事件传递给使用者。

new net.Socket([options])

创建一个新的 `socket` 对象。

`options` 是一个有以下默认值的对象：

```
{ fd: null
  allowHalfOpen: false,
  readable: false,
  writable: false
}
```

`fd` 允许你使用一个指定的已存在的 `socket` 文件描述符。设置 `readable` 和/或 `writable` 为 `true` 将允许从这个 `socket` 中读 和/或 写（注意，仅在传递了 `passed` 时可用）。关于 `allowHalfOpen` ，参阅 `createServer()` 和 `end` 事件。

socket.connect(options[, connectListener])

从给定的 `socket` 打开一个连接。

对于TCP `socket` ， `options` 参数需是一个包含以下属性的对象：

- port: 客户端需要连接的端口（必选）。
- host: 客户端需要连接的主机（默认：'localhost'）
- localAddress: 将要绑定的本地接口，为了网络连接。
- localPort: 将要绑定的本地端口，为了网络连接。
- family : IP协议族版本，默认为 `4` 。
- lookup : 自定义查找函数。默认为 `dns.lookup` 。

对于本地domain `socket` ， `options` 参数需是一个包含以下属性的对象：

- path: 客户端需要连接的路径（必选）。

通常这个方法是不需要的，因为通过 `net.createConnection` 打开 `socket` 。只有在你自定义了 `socket` 时才使用它。

这个函数式异步的，当 `connect` 事件触发时，这个 `socket` 就被建立了。如果在连接的过程有问题，那么 `connect` 事件将不会触发， `error` 将会带着这个异常触发。

`connectListener` 参数会被自动添加为 `connect` 事件的监听器。

socket.connect(port[, host][, connectListener])

socket.connect(path[, connectListener])

参阅 `socket.connect(options[, connectListener])` 。

socket.bufferSize

`net.Socket` 的属性，用于 `socket.write()` 。它可以帮助用户获取更快的运行速度。计算机不能一直保持大量数据被写入 `socket` 的状态，网络连接可以很慢。 `node.js` 在内部会排队等候数据被写入 `socekt` 并确保传输连接上的数据完好。(内部实现为：轮询 `socekt` 的文件描述符等待它为可写)。

内部缓存的可能结果是内存使用会增长。这个属性展示了缓存中还有多少待写入的字符（字符的数目约等于要被写入的字节数，但是缓冲区可能包含字符串，而字符串是惰性编码的，所以确切的字节数是未知的）。

遇到数值很大或增长很快的 `bufferSize` 时，应当尝试使用 `pause()` 和 `resume()` 来控制。

socket.setEncoding([encoding])

设置 `socket` 的编码作为一个可读流。详情参阅 `stream.setEncoding()` 。

socket.write(data[, encoding][, callback])

在套接字上发送数据。第二个参数指定了字符串的编码，默认为UTF8。

如果所有数据成功被刷新至了内核缓冲区，则返回 `true` 。如果所有或部分数据仍然在用户内存中排队，则返回 `false` 。 `drain` 事件将会被触发当 `buffer` 再次为空时。

当数据最终被写入时， `callback` 回调函数将会被执行，但可能不会马上执行。

socket.end([data][, encoding])

半关闭一个 `socket` 。比如，它发送一个 `FIN` 报文。可能服务器仍然在发送一些数据。

如果 `data` 参数被指定，那么等同于先调用 `socket.write(data, encoding)` ，再调用 `socket.end()` 。

socket.destroy()

确保这个 `socket` 上没有I/O活动发生。只在发生错误情况才需要（如处理错误）。

socket.pause()

暂停数据读取。 `data` 事件将不会再触发。对于控制上传非常有用。

socket.resume()

用于在调用 `pause()` 后，恢复数据读取。

socket.setTimeout(timeout[, callback])

如果 `socket` 在 `timeout` 毫秒中没有活动后，设置其为超时。默认情况下， `net.Socket` 没有超时。

当超时发生， `socket` 会收到一个 `timeout` 事件，但是连接将不会被断开。用户必须手动地调用 `end()` 或 `destroy()` 方法。

如果 `timeout` 是 `0` ，那么现有的超时将会被禁用。

可选的 `callback` 参数就会被自动添加为 `timeout` 事件的监听器。

返回一个 `socket` 。

socket.setNoDelay([noDelay])

警用纳格算法（Nagle algorithm）。默认情况下TCP连接使用纳格算法，它们的数据在被发送前会被缓存。设置 `noDelay` 为 `true` 将会在每次 `socket.write()` 时立刻发送数据。 `noDelay` 默认为 `true` 。

返回一个 `socket` 。

socket.setKeepAlive([enable][, initialDelay])

启用/警用长连接功能，并且在第一个在闲置 `socket` 的长连接 `probe` 被发送前，可选得设置初始延时。 `enable` 默认为 `false` 。

设定 `initialDelay` (毫秒)，来设定在收到的最后一个数据包和第一个长连接 `probe` 之间的延时。将 `initialDelay` 设成 `0` 会让值保持不变(默认值或之前所设的值)。默认为 `0` 。

返回一个 `socket` 。

socket.address()

返回绑定的地址，协议族名和端口通过操作系统报告。对查找操作系统分配的地址哪个端口被分配非常有用。返回一个有三个属性的对象。如 `{ port: 12346, family: 'IPv4', address: '127.0.0.1' }` 。

socket.unref()

调用一个 `socket` 对象的 `unref` 方法将允许如果它是事件系统中唯一活跃的 `socket` ，程序将会退出。如果 `socket` 已经被调用过这个方法，那么再次调用这个方法将不会有任何效果。

返回 `socket` 对象。

socket.ref()

与 `unref` 相反，在一个已经被调用 `unref` 方法的 `socket` 中调用 `ref` 方法，那么如果它是唯一活跃的 `socket` 时，程序将不会退出（默认）。如果 `socket` 已经被调用过这个方法，那么再次调用这个方法将不会有任何效果。

返回 `socket` 对象。

socket.remoteAddress

远程IP地址字符串。例如，`'74.125.127.100'` 或 `'2001:4860:a005::68'` 。

socket.remoteFamily

远程IP协议族字符串。例如，`'IPv4'` 或 `'IPv6'` 。

socket.remotePort

远程端口数值。例如，`80` 或 `21` 。

socket.localAddress

远程客户端正连接的本地IP地址字符串。例如，如果你正在监听 `'0.0.0.0'` 并且客户端连接在 `'192.168.1.1'` ，其值将为 `'192.168.1.1'` 。

socket.localPort

本地端口数值。例如，`80` 或 `21` 。

socket.bytesRead

接受的字节数。

socket.bytesWritten

发送的字节数。

net.Socket `net.Socket` 实例是一个包含以下事件的 `EventEmitter` ：

Event: 'lookup'

在解析主机名后，连接主机前触发。对UNIX `socket` 不适用。

- err {Error | Null} 错误对象，参阅 `dns.lookup()`
- address {String} IP地址
- family {String | Null} 地址类型。参阅'dns.lookup()`

Event: 'connect'

在 `socket` 连接成功建立后触发。参阅 `connect()` 。

Event: 'data'

- Buffer object

在接受到数据后触发。参数将会是一个 `Buffer` 或一个字符串。数据的编码由 `socket.setEncoding()` 设置（更多详细信息请查看可读流章节）。

注意，当 `socket` 触发 `data` 事件时，如果没有监听器存在。那么数据将会丢失。

Event: 'end'

当另一端的 `socket` 发送一个 `FIN` 报文时触发。

默认情况（ `allowHalfOpen == false` ）下，一旦一个 `socket` 的文件描述符被从它的等待写队列（pending write queue）中写出， `socket` 会销毁它。但是，当设定 `allowHalfOpen == true` 后， `socket` 不会在它这边自动调用 `end()` ，允许用户写入任意数量的数据，需要注意的是用户需要在自己这边调用 `end()` 。

Event: 'timeout'

当 `socket` 因不活动而超时时触发。这只是来表示 `socket` 被限制。用户必须手动关闭连接。

参阅 `socket.setTimeout()` 。

Event: 'drain'

当写缓冲为空时触发。可以被用来控制上传流量。

参阅 `socket.write()` 的返回值。

Event: 'error'

- Error object

当发生错误时触发。 `close` 事件会紧跟着这个事件触发。

Event: 'close'

- had_error 如果 `socket` 有一个传输错误时为 `true`

当 `socket` 完全关闭时触发。参数 `had_error` 是一个表示 `socket` 是否是因为传输错误而关闭的布尔值。

net.isIP(input)

测试 `input` 是否是一个IP地址。如果是不合法字符串时，会返回 `0` 。如果是IPv4地址则返回 `4` ，是IPv6地址则返回 `6` 。

net.isIPv4(input)

如果 `input` 是一个IPv4地址则返回 `true` ，否则返回 `false` 。

net.isIPv6(input)

如果 `input` 是一个IPv6地址则返回 `true` ，否则返回 `false` 。

OS

稳定度: 2 - 稳定

提供一些基本的操作系统相关的功能。

使用 `require('os')` 来获得这个模块。

os.tmpdir()

返回操作系统默认的临时文件目录。

os.homedir()

返回当前用户的家目录。

os.endianness()

返回CPU的字节序。 `BE` 为大端字节序， `LE` 为小端字节序。

os.hostname()

返回当前操作系统的主机名。

os.type()

返回操作系统名。例如，Linux下为 `'Linux'` ，OS X下为 `'Darwin'` ，Windows下为 `'Windows_NT'` 。

os.platform()

返回操作系统平台。可能的值有 `'darwin'` ， `'freebsd'` ， `'linux'` ， `'sunos'` 或 `'win32'` 。返回 `process.platform` 值。

os.arch()

返回操作系统CPU架构。可能的值有 `'x64'` ，`'arm'` 和 `'ia32'` 。返回 `process.arch` 值。

os.release()

返回操作系统的发行版本。

os.uptime()

返回操作系统的运行时间（秒）。

os.loadavg()

返回一个包含1，5，15分钟平均负载的数组。

平均负载是一个系统活动测量，由操作系统计算并且由一个分数表示。根据经验，理想的负载均衡数应该比系统的逻辑CPU数小。

平均负载完全是一个 `UNIX-y` 概念；在Windows中没有完全对等的概念。所以在Windows下，这个函数总是返回 `[0, 0, 0]` 。

os.totalmem()

以字节的形式返回系统的总内存。

os.freemem()

以字节的形式返回系统的可用内存。

os.cpus()

返回一个包含安装的各个CPU/核心信息的对象数组：型号，速度（单位MHz），和时间（一个包含CPU/核心花费的毫秒数的对象： `user` ，`nice` ，`sys` ，`idle` 和 `irq` ）。

`os.cpus` 例子：

```
[ { model: 'Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz',
    speed: 2926,
    times:
```

```
      { user: 252020,
        nice: 0,
        sys: 30340,
        idle: 1070356870,
        irq: 0 } },
  { model: 'Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz',
    speed: 2926,
    times:
     { user: 306960,
       nice: 0,
       sys: 26980,
       idle: 1071569080,
       irq: 0 } },
  { model: 'Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz',
    speed: 2926,
    times:
     { user: 248450,
       nice: 0,
       sys: 21750,
       idle: 1070919370,
       irq: 0 } },
  { model: 'Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz',
    speed: 2926,
    times:
     { user: 256880,
       nice: 0,
       sys: 19430,
       idle: 1070905480,
       irq: 20 } },
  { model: 'Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz',
    speed: 2926,
    times:
     { user: 511580,
       nice: 20,
       sys: 40900,
       idle: 1070842510,
       irq: 0 } },
  { model: 'Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz',
    speed: 2926,
    times:
```

```
      { user: 291660,
        nice: 0,
        sys: 34360,
        idle: 1070888000,
        irq: 10 } },
   { model: 'Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz',
     speed: 2926,
     times:
      { user: 308260,
        nice: 0,
        sys: 55410,
        idle: 1071129970,
        irq: 880 } },
   { model: 'Intel(R) Core(TM) i7 CPU         860  @ 2.80GHz',
     speed: 2926,
     times:
      { user: 266450,
        nice: 1480,
        sys: 34920,
        idle: 1072572010,
        irq: 30 } } ]
```

注意因为 `nice` 值是UNIX中心的，所以在Windows中所有进程的 `nice` 值都将是 `0` 。

os.networkInterfaces()

获取一个网络接口列表：

```
{ lo:
   [ { address: '127.0.0.1',
       netmask: '255.0.0.0',
       family: 'IPv4',
       mac: '00:00:00:00:00:00',
       internal: true },
     { address: '::1',
       netmask: 'ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff',
       family: 'IPv6',
       mac: '00:00:00:00:00:00',
       internal: true } ],
  eth0:
   [ { address: '192.168.1.108',
       netmask: '255.255.255.0',
       family: 'IPv4',
       mac: '01:02:03:0a:0b:0c',
       internal: false },
     { address: 'fe80::a00:27ff:fe4e:66a1',
       netmask: 'ffff:ffff:ffff:ffff::',
       family: 'IPv6',
       mac: '01:02:03:0a:0b:0c',
       internal: false } ] }
```

注意，由于底层系统实现的原因，它将只会返回被赋予一个地址的网络接口。

os.EOL

一个定义了对于操作系统，合适的行结束记号的常量。

Path

稳定度: 2 - 稳定

这个模块提供了处理和转换文件路径的工具。几乎所有的方法都仅提供字符串转换功能。文件系统不会去检查路径是否可用。

通过 `require('path')` 来使用这个模块。以下是提供的方法：

path.normalize(p)

规范化字符串路径，注意 `'..'` 和 `'.'` 部分。

当有多个连续斜杠时，它们会被替换为一个斜杠；当路径的最后有一个斜杠，它会被保留。在Windows下使用反斜杠。

例子：

```
path.normalize('/foo/bar//baz/asdf/quux/..')
// returns
'/foo/bar/baz/asdf'
```

path.join([path1][, path2][, ...])

连接所有的参数，并且规范化结果路径。

参数必须是字符串。在0.8版本中，非字符串参数会被忽略，在0.10版本及之后，会抛出一个异常。

例子：

```
path.join('/foo', 'bar', 'baz/asdf', 'quux', '..')
// returns
'/foo/bar/baz/asdf'

path.join('foo', {}, 'bar')
// throws exception
TypeError: Arguments to path.join must be strings
```

path.resolve([from ...], to)

将 `to` 解析为绝对路径。

如果 `to` 不已经是相对于 `from` 参数的绝对路径， `to` 会被添加到 `from` 的右边，直到找出绝对了路径。如果使用了 `from` 中所有的路径仍没有找出绝对路径，当前的工作路径也会被使用。结果路径会被规范化，并且结尾的斜杠会被移除，除非解析得到了一个根路径。非字符串参数会被忽略。

另一个思路是将它看做shell中一系列的 `cd` 命令：

```
path.resolve('foo/bar', '/tmp/file/', '..', 'a/../subfile')
```

相似于：

```
cd foo/bar
cd /tmp/file/
cd ..
cd a/../subfile
pwd
```

区别是不同的路径不需要一定存在，并且可以是文件。

例子：

```
path.resolve('/foo/bar', './baz')
// returns
'/foo/bar/baz'

path.resolve('/foo/bar', '/tmp/file/')
// returns
'/tmp/file'

path.resolve('wwwroot', 'static_files/png/', '../gif/image.gif')
// if currently in /home/myself/iojs, it returns
'/home/myself/iojs/wwwroot/static_files/gif/image.gif'
```

path.isAbsolute(path)

判断 `path` 是否是一个绝对路径。一个绝对路径总是被解析为相同的路径，无论当前工作目录是哪里。

Posix例子：

```
path.isAbsolute('/foo/bar') // true
path.isAbsolute('/baz/..')  // true
path.isAbsolute('qux/')     // false
path.isAbsolute('.')        // false
```

Windows例子：

```
path.isAbsolute('//server')  // true
path.isAbsolute('C:/foo/..') // true
path.isAbsolute('bar\\baz')   // false
path.isAbsolute('.')         // false
```

path.relative(from, to)

解析从 `from` 到 `to` 的相对路径。

当我们有两个绝对路径，并且我们要得到它们间一个对于另外一个的相对路径。这实际上是 `path.resolve` 的相反操作。我们可以看看这是什么意思：

```
path.resolve(from, path.relative(from, to)) == path.resolve(to)
```

例子：

```
path.relative('C:\\orandea\\test\\aaa', 'C:\\orandea\\impl\\bbb'
)
// returns
'..\\..\\impl\\bbb'

path.relative('/data/orandea/test/aaa', '/data/orandea/impl/bbb'
)
// returns
'../../impl/bbb'
```

path.dirname(p)

返回路径的目录名。与Unix `dirname` 命令相似。

例子：

```
path.dirname('/foo/bar/baz/asdf/quux')
// returns
'/foo/bar/baz/asdf'
```

path.basename(p[, ext])

返回路径中的最后一部分。与Unix `basename` 命令相似。

例子：

```
path.basename('/foo/bar/baz/asdf/quux.html')
// returns
'quux.html'

path.basename('/foo/bar/baz/asdf/quux.html', '.html')
// returns
'quux'
```

path.extname(p)

返回路径的扩展名，即从路径的最后一部分中的最后一个 `'.'` 到末尾之间的字符串。如果路径的最后一部分没有 `'.'` ，或者第一个字符是 `'.'` ，那么将返回一个空字符串，例子：

```
path.extname('index.html')
// returns
'.html'

path.extname('index.coffee.md')
// returns
'.md'

path.extname('index.')
// returns
'.'

path.extname('index')
// returns
''
```

path.sep

返回特定平台的文件分隔符。 `'\\'` 或 `'/'` 。

一个*nix上的例子：

```
'foo/bar/baz'.split(path.sep)
// returns
['foo', 'bar', 'baz']
```

一个Windows上的例子：

```
'foo\\bar\\baz'.split(path.sep)
// returns
['foo', 'bar', 'baz']
```

path.delimiter

特定平台的路径分隔符， `';'` 或 `':'` 。

一个*nix上的例子：

```
console.log(process.env.PATH)
// '/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin'

process.env.PATH.split(path.delimiter)
// returns
['/usr/bin', '/bin', '/usr/sbin', '/sbin', '/usr/local/bin']
```

一个Windows上的例子：

```
console.log(process.env.PATH)
// 'C:\Windows\system32;C:\Windows;C:\Program Files\iojs\'

process.env.PATH.split(path.delimiter)
// returns
['C:\\Windows\\system32', 'C:\\Windows', 'C:\\Program Files\\ioj
s\\']
```

path.parse(pathString)

根据一个路径字符串返回一个对象。

一个*nix上的例子：

```
path.parse('/home/user/dir/file.txt')
// returns
{
    root : "/",
    dir : "/home/user/dir",
    base : "file.txt",
    ext : ".txt",
    name : "file"
}
```

一个Windows上的例子：

```
path.parse('C:\\path\\dir\\index.html')
// returns
{
    root : "C:\\",
    dir : "C:\\path\\dir",
    base : "index.html",
    ext : ".html",
    name : "index"
}
```

path.format(pathObject)

根据一个对象，返回一个路径字符串，与 `path.parse` 相反。

```
path.format({
    root : "/",
    dir : "/home/user/dir",
    base : "file.txt",
    ext : ".txt",
    name : "file"
})
// returns
'/home/user/dir/file.txt'
```

path.posix

提供对上述的路径方法的访问，但是总是以兼容posix的方式交互（interact）。

path.win32

提供对上述的路径方法的访问，但是总是以兼容win32的方式交互（interact）。

process

`process` 对象是一个全局对象，并且何以被在任何地方调用。这是一个 `EventEmitter` 实例。

Exit Codes

当没有任何异步操作在等待时， `node.js` 通常将会以一个0为退出码退出。以下这些状态码会在其他情况下被用到：

- 1 未捕获的致命异常。这是一个未捕获的异常，并且它没有被 `domain` 处理，也没有被 `uncaughtException` 处理。
- 2 未使用（由 `Bash` 为内建误操作保留）。
- 3 内部的 `JavaScript` 解析错误。 `node.js` 内部的 `JavaScript` 源码引导（bootstrapping）造成的一个解释错误。这极其罕见。并且常常只会发生在 `node.js` 自身的开发过程中。
- 4 内部的 `JavaScript` 求值错误。 `node.js` 内部的 `JavaScript` 源码引导（bootstrapping）未能在求值时返回一个函数值。这极其罕见。并且常常只会发生在 `node.js` 自身的开发过程中。
- 5 致命错误。这是V8中严重的不可恢复的错误。典型情况下，一个带有 `FATAL ERROR` 前缀的信息会被打印在 `stderr` 。
- 6 内部异常处理函数丧失功能。这是一个未捕获异常，但是内部的致命异常处理函数被设置为丧失功能，并且不能被调用。
- 7 内部异常处理函数运行时失败。这是一个未捕获异常，并且内部致命异常处理函数试图处理它时，自身抛出了一个错误。例如它可能在当 `process.on('uncaughtException')` 或 `domain.on('error')` 处理函数抛出错误时发生。
- 8 未使用。 `node.js` 的之前版本中，退出码 `8` 通常表示一个未捕获异常。
- 9 无效参数。当一个位置的选项被指定，或者一个必选的值没有被提供。
- 10 内部的 `JavaScript` 运行时错误。 `node.js` 内部的 `JavaScript` 源码引导（bootstrapping）函数被调用时抛出一个错误。这极其罕见。并且常常只会发生在 `node.js` 自身的开发过程中。
- 12 无效的调试参数。 `--debug` 和/或 `--debug-brk` 选项被设置，当时选择了一个无效的端口。
- 大于128 信号退出。如果 `node.js` 收到了一个如 `SIGKILL` 或 `SIGHUP` 的致

> 命信号，那么它将以一个 `128` 加上 信号码的值 的退出码退出。这是一个标准的Unix实践，因为退出码由一个7位整数定义，并且信号的退出设置了一个高顺序位（high-order bit），然后包含一个信号码的值。

Event: 'exit'

进程即将退出时触发。在这个时刻已经没有办法可以阻止事件循环的退出，并且一旦所有的 `exit` 监听器运行结束时，进程将会退出。因此，在这个监听器中你仅仅能调用同步的操作。这是检查模块状态（如单元测试）的好钩子。回调函数有一个退出码参数。

例子：

```
process.on('exit', function(code) {
  // do *NOT* do this
  setTimeout(function() {
    console.log('This will not run');
  }, 0);
  console.log('About to exit with code:', code);
});
```

Event: 'beforeExit'

这个事件在 `node.js` 清空了它的事件循环并且没有任何已安排的任务时触发。通常 `node.js` 当没有更多被安排的任务时就会退出，但是 `beforeExit` 中可以执行异步调用，让 `node.js` 继续运行。

`beforeExit` 在程序被显示终止时不会触发，如 `process.exit()` 或未捕获的异常。除非想去安排更多的任务，否则它不应被用来做为 `exit` 事件的替代。

Event: 'uncaughtException'

当一个异常冒泡回事件循环时就会触发。如果这个时间被添加了监听器，那么默认行为（退出程序且打印堆栈跟踪信息）将不会发生。

例子：

```
process.on('uncaughtException', function(err) {
  console.log('Caught exception: ' + err);
});

setTimeout(function() {
  console.log('This will still run.');
}, 500);

// Intentionally cause an exception, but don't catch it.
nonexistentFunc();
console.log('This will not run.');
```

注意， `uncaughtException` 来处理异常是非常粗糙的。

请不要使用它，使用 `domain` 来替代。如果你已经使用了它，请在不处理这个异常之后重启你的应用。

请不要像 `node.js` 的 `Error Resume Next` 这样使用。一个未捕获异常意味着你的应用或拓展有未定义的状态。盲目地恢复意味着任何事都可能发生。

想象你在升级你的系统时电源被拉断了。10次中前9次都没有问题，但是第10次时，你的系统崩溃了。

你已经被警告。

Event: 'unhandledRejection'

在一个事件循环中，当一个 `promise` 被“拒绝”并且没有附属的错误处理函数时触发。当一个带有 `promise` 异常的程序被封装为被“拒绝”的 `promise` 时，这样的程序的错误可以被 `promise.catch(...)` 捕获处理并且“拒绝”会通过 `promise` 链冒泡。这个事件对于侦测和保持追踪那些“拒绝”没有被处理的 `promise` 非常有用。这个事件会带着以下参数触发：

- reason `promise` 的“拒绝”对象（通常是一个错误实例）
- p 被“拒绝”的 `promise`

下面是一个把所有未处理的“拒绝”打印到控制台的例子：

```
process.on('unhandledRejection', function(reason, p) {
    console.log("Unhandled Rejection at: Promise ", p, " reason: ", reason);
    // application specific logging, throwing an error, or other logic here
});
```

下面是一个会触发 `unhandledRejection` 事件的“拒绝”：

```
somePromise.then(function(res) {
  return reportToUser(JSON.pasre(res)); // note the typo
}); // no `.catch` or `.then`
```

Event: 'rejectionHandled'

当一个 `Promise` 被“拒绝”并且一个错误处理函数被附给了它（如 `.catch()` ）时的下一个事件循环之后触发。这个事件会带着以下参数触发：

- p 一个在之前会被触发在 `unhandledRejection` 事件中，但现在被处理函数捕获的 `promise`

一个 `promise` 链的顶端没有 “拒绝”可以总是被处理 的概念。由于其异步的本质，一个 `promise` 的“拒绝”可以在未来的某一个时间点被处理，可以是在事件循环中被触发 `unhandledRejection` 事件之后。

另外，不像同步代码中是一个永远增长的 未捕获异常 列表， `promise` 中它是一个可伸缩的 未捕获 `拒绝` 列表。在同步代码中， `uncaughtException` 事件告诉你 未捕获异常 列表增长了。但是在 `promise` 中， `unhandledRejection` 事件告诉你 未捕获“拒绝” 列表增长了， `rejectionHandled` 事件告诉你 未捕获“拒绝” 列表缩短了。

使用“拒绝”侦测钩子来保持一个被“拒绝”的 `promise` 列表：

```
var unhandledRejections = [];
process.on('unhandledRejection', function(reason, p) {
    unhandledRejections.push(p);
});
process.on('rejectionHandled', function(p) {
    var index = unhandledRejections.indexOf(p);
    unhandledRejections.splice(index, 1);
});
```

Signal Events

当一个进程收到一个信号时触发。参阅 `sigaction(2)` 。

监听 `SIGINT` 信号的例子：

```
// Start reading from stdin so we don't exit.
process.stdin.resume();

process.on('SIGINT', function() {
  console.log('Got SIGINT.  Press Control-D to exit.');
});
```

一个发送 `SIGINT` 信号的快捷方法是在大多数终端中按下 Control-C 。

注意：

- SIGUSR1 是 `node.js` 用于开启调试的保留信号。可以为其添加一个监听器，但不能阻止调试的开始。
- SIGTERM 和 SIGINT在非Windows平台下有在以 128 + 信号 退出码退出前重置终端模式的默认监听器。如果另有监听器被添加，默认监听器会被移除（即 `node.js` 将会不再退出）。
- SIGPIPE 默认被忽略，可以被添加监听器。
- SIGHUP 当控制台被关闭时会在Windows中产生，或者其他平台有其他相似情况时（参阅 `signal(7)` ）。它可以被添加监听器，但是Windows中 `node.js` 会无条件的在10秒后关闭终端。在其他非Windows平台，它的默认行为是结束 `node.js` ，但是一旦被添加了监听器，默认行为会被移除。
- SIGTERM 在Windows中不被支持，它可以被监听。
- SIGINT 支持所有的平台。可以由 CTRL+C 产生（尽管它可能是可配置的）。

当启用终端的 `raw mode` 时，它不会产生。
- SIGBREAK 在Windows中，按下 CTRL+BREAK 时它会产生。在非Windows平台下，它可以被监听，但它没有产生的途径。
- SIGWINCH 当终端被改变大小时产生。Windows下，它只会在当光标被移动时写入控制台或可读tty使用 `raw mode` 时发生。
- SIGKILL 可以被添加监听器。它会无条件得在所有平台下关闭 `node.js` 。
- SIGSTOP 可以被添加监听器。

注意Windows不支持发送信号，但 `node.js` 通过 `process.kill()` 和 `child_process.kill()` 提供了模拟：- 发送信号 `0` 被用来检查进程的存在 - 发送SIGINT， SIGTERM 和 SIGKILL 会导致目标进程的无条件退出。

process.stdout

一个指向 `stdout` 的可写流。

例如， `console.log` 可能与这个相似：

```
console.log = function(msg) {
  process.stdout.write(msg + '\n');
};
```

在 `node.js` 中， `process.stderr` 和 `process.stdout` 与其他流不同，因为他们不能被关闭（调用 `end()` 会报错）。它们永远不触发 `finish` 事件并且写操作通常是阻塞的。

- 当指向普通文件或TTY文件描述符时，它们是阻塞的。
- 以下情况下他们指向流
  - 他们在Linux/Unix中阻塞
  - 他们在Windows中的其他流里不阻塞

若要检查 `node.js` 是否在一个TTY上下文中运行，读取 `process.stderr` ， `process.stdout` 或 `process.stdin` 的 `isTTY` 属性：

```
$ iojs -p "Boolean(process.stdin.isTTY)"
true
$ echo "foo" | iojs -p "Boolean(process.stdin.isTTY)"
false

$ iojs -p "Boolean(process.stdout.isTTY)"
true
$ iojs -p "Boolean(process.stdout.isTTY)" | cat
false
```

更多信息请参阅tty文档。

process.stderr

一个指向 `stderr` 的可写流。

在 `node.js` 中， `process.stderr` 和 `process.stdout` 与其他流不同，因为他们不能被关闭（调用 `end()` 会报错）。它们永远不触发 `finish` 事件并且写操作通常是阻塞的。

- 当指向普通文件或TTY文件描述符时，它们是阻塞的。
- 以下情况下他们指向流
    - 他们在Linux/Unix中阻塞
    - 他们在Windows中的其他流里不阻塞

process.stdin

一个指向 `stdin` 的可读流。

一个打开标准输入并且监听两个事件的例子：

```
process.stdin.setEncoding('utf8');

process.stdin.on('readable', function() {
  var chunk = process.stdin.read();
  if (chunk !== null) {
    process.stdout.write('data: ' + chunk);
  }
});

process.stdin.on('end', function() {
  process.stdout.write('end');
});
```

作为一个流， `process.stdin` 可以被切换至“旧”模式，这样就可以兼容 `node.js` v0.10 前所写的脚本。更多信息请参阅 流的兼容性 。

在“旧”模式中 `stdin` 流默认是被暂停的。所以你必须调用 `process.stdin.resume()` 来读取。注意调用 `process.stdin.resume()` 这个操作本身也会将流切换至旧模式。

如果你正将开启一个新的工程。你应该要更常使用“新”模式的流。

process.argv

一个包含了命令行参数的数组。第一次元素将会是 `'iojs'` ，第二个元素将会是 `JavaScript` 文件名。之后的元素将会是额外的命令行参数。

```
// print process.argv
process.argv.forEach(function(val, index, array) {
  console.log(index + ': ' + val);
});
```

这将会是：

```
$ iojs process-2.js one two=three four
0: iojs
1: /Users/mjr/work/iojs/process-2.js
2: one
3: two=three
4: four
```

process.execPath

这将是开启进程的可执行文件的绝对路径名：

例子：

```
/usr/local/bin/iojs
```

process.execArgv

这是在启动时 `node.js` 自身参数的集合。这些参数不会出现在 `process.argv` 中，并且不会包含 `node.js` 可执行文件，脚本名和其他脚本名之后的参数。这些参数对开启和父进程相同执行环境的子进程非常有用。

例子：

```
$ iojs --harmony script.js --version
```

`process.execArgv` 将会是：

```
['--harmony']
```

`process.argv` 将会是：

```
['/usr/local/bin/iojs', 'script.js', '--version']
```

process.abort()

这将导致 `node.js` 触发 `abort` 事件。这个将导致 `node.js` 退出，并创建一个核心文件。

process.chdir(directory)

为进程改变当前工作目录，如果失败，则抛出一个异常。

```
console.log('Starting directory: ' + process.cwd());
try {
  process.chdir('/tmp');
  console.log('New directory: ' + process.cwd());
}
catch (err) {
  console.log('chdir: ' + err);
}
```

process.cwd()

返回进程的当前工作目录。

```
console.log('Current directory: ' + process.cwd());
```

process.env

包含用户环境变量的对象。参阅 `environ(7)` 。

一个例子：

```
{ TERM: 'xterm-256color',
  SHELL: '/usr/local/bin/bash',
  USER: 'maciej',
  PATH: '~/.bin/:/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin',
  PWD: '/Users/maciej',
  EDITOR: 'vim',
  SHLVL: '1',
  HOME: '/Users/maciej',
  LOGNAME: 'maciej',
  _: '/usr/local/bin/iojs' }
```

你可以改写这个对象，但是改变不会反应在你的进程之外。这以为着以下代码不会正常工作：

```
$ iojs -e 'process.env.foo = "bar"' && echo $foo
```

但是以下代码会：

```
process.env.foo = 'bar';
console.log(process.env.foo);
```

process.exit([code])

使用指定的退出码退出程序，如果忽略退出码。那么将使用“成功”退出码 `0` 。

以一个“失败”退出码结束：

```
process.exit(1);
```

在执行 `node.js` 的shell中可以看到为 `1` 的退出码。

process.exitCode

将是程序退出码的数字，当程序优雅退出 或 被 `process.exit()` 关闭且没有指定退出码时。

为 `process.exit(code)` 指定一个退出码会覆盖之前的 `process.exitCode` 设置。

process.getgid()

注意：这个函数只在POSIX平台上有效（如在Windows，Android中无效）。

获取进程的群组标识（参阅 `getgid(2)` ）。这是一个群组id数组，不是群组名。

```
if (process.getgid) {
  console.log('Current gid: ' + process.getgid());
}
```

process.getegid()

注意：这个函数只在POSIX平台上有效（如在Windows，Android中无效）。

获取进程的有效群组标识（参阅 `getgid(2)` ）。这是一个群组id数组，不是群组名。

```
if (process.getegid) {
  console.log('Current gid: ' + process.getegid());
}
```

process.setgid(id)

注意：这个函数只在POSIX平台上有效（如在Windows，Android中无效）。

设置进程的群组标识（参阅 `setgid(2)` ）。它接受一个数字ID或一个群组名字符串。如果群组名被指定，那么这个方法将在解析群组名为一个ID的过程中阻塞。

```
if (process.getgid && process.setgid) {
  console.log('Current gid: ' + process.getgid());
  try {
    process.setgid(501);
    console.log('New gid: ' + process.getgid());
  }
  catch (err) {
    console.log('Failed to set gid: ' + err);
  }
}
```

process.setegid(id)

注意：这个函数只在POSIX平台上有效（如在Windows，Android中无效）。

设置进程的有效群组标识（参阅 `setgid(2)` ）。它接受一个数字ID或一个群组名字符串。如果群组名被指定，那么这个方法将在解析群组名为一个ID的过程中阻塞。

```
if (process.getegid && process.setegid) {
  console.log('Current gid: ' + process.getegid());
  try {
    process.setegid(501);
    console.log('New gid: ' + process.getegid());
  }
  catch (err) {
    console.log('Failed to set gid: ' + err);
  }
}
```

process.getuid()

注意：这个函数只在POSIX平台上有效（如在Windows，Android中无效）。

获取进程的用户id（参阅 `getuid(2)` ）。这是一个数字用户id，不是用户名。

```
if (process.getuid) {
  console.log('Current uid: ' + process.getuid());
}
```

process.geteuid()

注意：这个函数只在POSIX平台上有效（如在Windows，Android中无效）。

获取进程的有效用户id（参阅 `getuid(2)` ）。这是一个数字用户id，不是用户名。

```
if (process.geteuid) {
  console.log('Current uid: ' + process.geteuid());
}
```

process.setuid(id)

注意：这个函数只在POSIX平台上有效（如在Windows，Android中无效）。

设置进程的用户ID（参阅 `setuid(2)` ）。它接受一个数字ID或一个用户名字符串。如果用户名被指定，那么这个方法将在解析用户名为一个ID的过程中阻塞。

```
if (process.getuid && process.setuid) {
  console.log('Current uid: ' + process.getuid());
  try {
    process.setuid(501);
    console.log('New uid: ' + process.getuid());
  }
  catch (err) {
    console.log('Failed to set uid: ' + err);
  }
}
```

process.seteuid(id)

注意：这个函数只在POSIX平台上有效（如在Windows，Android中无效）。

设置进程的有效用户ID（参阅 `seteuid(2)` ）。它接受一个数字ID或一个用户名字符串。如果用户名被指定，那么这个方法将在解析用户名为一个ID的过程中阻塞。

```
if (process.geteuid && process.seteuid) {
  console.log('Current uid: ' + process.geteuid());
  try {
    process.seteuid(501);
    console.log('New uid: ' + process.geteuid());
  }
  catch (err) {
    console.log('Failed to set uid: ' + err);
  }
}
```

process.getgroups()

注意：这个函数只在POSIX平台上有效（如在Windows，Android中无效）。

返回一个补充群组ID的数组。如果包含了有效的组ID，POSIX将不会指定。但 `node.js` 保证它始终是。

process.setgroups(groups)

注意：这个函数只在POSIX平台上有效（如在Windows，Android中无效）。

设置一个补充群组ID。这是一个特殊的操作，意味着你需要拥有 `root` 或 `CAP_SETGID` 权限才可以这么做。

列表可以包含群组ID，群组名，或两者。

process.initgroups(user, extra_group)

注意：这个函数只在POSIX平台上有效（如在Windows，Android中无效）。

读取 `/etc/group` 并且初始化群组访问列表，使用用户是组员的所有群组。这是一个特殊的操作，意味着你需要拥有 `root` 或 `CAP_SETGID` 权限才可以这么做。

`user` 是一个用户名或一个用户ID。 `extra_group` 是一个群组名或群组ID。

当你注销权限时有些需要关心的：

```
console.log(process.getgroups());         // [ 0 ]
process.initgroups('bnoordhuis', 1000);   // switch user
console.log(process.getgroups());         // [ 27, 30, 46, 1000, 0 ]
process.setgid(1000);                     // drop root gid
console.log(process.getgroups());         // [ 27, 30, 46, 1000 ]
```

process.version

一个暴露 `NODE_VERSION` 的编译时存储属性。

```
console.log('Version: ' + process.version);
```

process.versions

一个暴露node.js版本和它的依赖的字符串属性。

```
console.log(process.versions);
```

将可能打印：

```
{ http_parser: '2.3.0',
  node: '1.1.1',
  v8: '4.1.0.14',
  uv: '1.3.0',
  zlib: '1.2.8',
  ares: '1.10.0-DEV',
  modules: '43',
  openssl: '1.0.1k' }
```

process.config

一个表示用于编译当前 `node.js` 执行文件的配置的JavaScript对象。这和运行 `./configure` 脚本产生的 `config.gypi` 一样。

一个可能的输出：

```
{ target_defaults:
   { cflags: [],
     default_configuration: 'Release',
     defines: [],
     include_dirs: [],
     libraries: [] },
  variables:
   { host_arch: 'x64',
     node_install_npm: 'true',
     node_prefix: '',
     node_shared_cares: 'false',
     node_shared_http_parser: 'false',
     node_shared_libuv: 'false',
     node_shared_zlib: 'false',
     node_use_dtrace: 'false',
     node_use_openssl: 'true',
     node_shared_openssl: 'false',
     strict_aliasing: 'true',
     target_arch: 'x64',
     v8_use_snapshot: 'true' } }
```

process.kill(pid[, signal])

给进程传递一个信号。 `pid` 是进程id， `signal` 是描述信号的字符串。信号码类似于 `'SIGINT'` 或 `'SIGHUP'` 。如果忽略，那么信号将是 `'SIGTERM'` 。更多信息参阅 `Signal Events` 和 `kill(2)` 。

如果目标不存在将会抛出一个错误，并且在一些情况下， `0` 信号可以被用来测试进程的存在。

注意，这个函数仅仅是名字为 `process.kill` ，它只是一个信号发送者。发送的信号可能与杀死进程无关。

一个发送信号给自身的例子：

```
process.on('SIGHUP', function() {
  console.log('Got SIGHUP signal.');
});

setTimeout(function() {
  console.log('Exiting.');
  process.exit(0);
}, 100);
```

process.kill(process.pid, 'SIGHUP');

注意：当 `SIGUSR1` 被 `node.js` 收到，它会开始调试。参阅 `Signal Events` 。

process.pid#

进程的PID。

```
console.log('This process is pid ' + process.pid);
```

process.title#

设置/获取 `'ps'` 中显示的进程名。

当设置该属性时，所能设置的字符串最大长度视具体平台而定，如果超过的话会自动截断。

在 Linux 和 OS X 上，它受限于名称的字节长度加上命令行参数的长度，因为它有覆盖参数内存。

v0.8 版本允许更长的进程标题字符串，也支持覆盖环境内存，但是存在潜在的不安全和混乱。

process.arch

返回当前的处理器结构：`'arm'`，`'ia32'` 或 `'x64'`。

```
console.log('This processor architecture is ' + process.arch);
```

process.platform

放回当前的平台：`'darwin'`，`'freebsd'`，`'linux'`，`'sunos'` 或 `'win32'`。

```
console.log('This platform is ' + process.platform);
```

process.memoryUsage()

返回当前 `node.js` 进程内存使用情况（用字节描述）的对象。

```
var util = require('util');

console.log(util.inspect(process.memoryUsage()));
```

可能的输出：

```
{ rss: 4935680,
  heapTotal: 1826816,
  heapUsed: 650472 }
```

`heapTotal` 和 `heapUsed` 指向V8的内存使用。

process.nextTick(callback[, arg][, ...])

- callback Function

在事件循环的下一次循环中调用回调函数。

这不是 `setTimeout(fn, 0)` 的简单别名，它更有效率。在之后的 `tick` 中，它在任何其他的I/O事件（包括 `timer` ）触发之前运行。

```
console.log('start');
process.nextTick(function() {
  console.log('nextTick callback');
});
console.log('scheduled');
// Output:
// start
// scheduled
// nextTick callback
```

这对于开发你想要给予用户在对象被构建后，任何I/O发生前，去设置事件监听器的机会时，非常有用。

```
function MyThing(options) {
  this.setupOptions(options);

  process.nextTick(function() {
    this.startDoingStuff();
  }.bind(this));
}

var thing = new MyThing();
thing.getReadyForStuff();
// thing.startDoingStuff() gets called now, not before.
```

这对于100%同步或100%异步的API非常重要。考虑一下例子：

```
// WARNING!  DO NOT USE!  BAD UNSAFE HAZARD!
function maybeSync(arg, cb) {
  if (arg) {
    cb();
    return;
  }

  fs.stat('file', cb);
}
```

这个API是危险的，如果你这样做：

```
maybeSync(true, function() {
  foo();
});
bar();
```

`foo()` 和 `bar()` 的调用次序是不确定的。

更好的做法是：

```
function definitelyAsync(arg, cb) {
  if (arg) {
    process.nextTick(cb);
    return;
  }

  fs.stat('file', cb);
}
```

注意： `nextTick` 队列在每一次事件循环的I/O开始前都要完全执行完毕。所以，递归地设置 `nextTick` 回调会阻塞I/O的方法，就像一个 `while(true);` 循环。

process.umask([mask])

设置或读取进程的文件模式的创建掩码。子进程从父进程中继承这个掩码。返回旧的掩码如果 `mask` 参数被指定。否则，会返回当前掩码。

```
var oldmask, newmask = 0022;

oldmask = process.umask(newmask);
console.log('Changed umask from: ' + oldmask.toString(8) +
            ' to ' + newmask.toString(8));
```

process.uptime()

`node.js` 进程已执行的秒数。

process.hrtime()

以 `[seconds, nanoseconds]` 元组数组的形式返回高分辨时间。是相对于过去的任意时间。它与日期无关所以不用考虑时区等因素。它的主要用途是衡量程序性能。

你可以将之前的 `process.hrtime()` 返回传递给一个新的 `process.hrtime()` 来获得一个比较。衡量性能时非常有用：

```
var time = process.hrtime();
// [ 1800216, 25 ]

setTimeout(function() {
  var diff = process.hrtime(time);
  // [ 1, 552 ]

  console.log('benchmark took %d nanoseconds', diff[0] * 1e9 + d
iff[1]);
  // benchmark took 1000000527 nanoseconds
}, 1000);
```

process.mainModule

检索 `require.main` 的备用方式。区别是，如果主模块在运行时改变， `require.main` 可能仍指向改变发生前的被引入的原主模块。通常，假设它们一样是安全的。

与 `require.main` 一样，当如果没有入口脚本时，它将是 `undefined` 。

punycode

Stability: 2 - Stable

`Punycode.js` 在 `node.js` v1.0.0+ 和 `Node.js` v0.6.2+ 中被内置。通过 `require('punycode')` 来获取它（若要在其他版本的 `node.js` 中使用它，需要先通过npm来安装 `punycode` 模块）。

punycode.decode(string)

转换一个纯ASCII符号 `Punycode` 字符串为一个 `Unicode` 符号的字符串。

```
// decode domain name parts
punycode.decode('maana-pta'); // 'mañana'
punycode.decode('--dqo34k'); // '☃-⌘'
```

punycode.encode(string)

转换一个 `Unicode` 符号的字符串为一个纯ASCII符号 `Punycode` 字符串。

```
// encode domain name parts
punycode.encode('mañana'); // 'maana-pta'
punycode.encode('☃-⌘'); // '--dqo34k'
```

punycode.toUnicode(domain)

转换一个代表了一个域名的 `Punycode` 字符串为一个 `Unicode` 字符串。只有代表了域名的部分的 `Punycode` 字符串会被转换。也就是说，如果你调用了一个已经被转换为 `Unicode` 的字符串，也是没有问题的。

```
// decode domain names
punycode.toUnicode('xn--maana-pta.com'); // 'mañana.com'
punycode.toUnicode('xn----dqo34k.com'); // '☃-⌘.com'
```

punycode.toASCII(domain)

转换一个代表了一个域名的 `Unicode` 字符串为一个 `Unicode` 字符串。只有代表了域名的部分的非ASCII字符串会被转换。也就是说，如果你调用了一个已经被转换为ASCII的字符串，也是没有问题的。

```
// encode domain names
punycode.toASCII('mañana.com'); // 'xn--maana-pta.com'
punycode.toASCII('☃-⌘.com'); // 'xn----dqo34k.com'
```

punycode.ucs2

punycode.ucs2.decode(string)

创建一个包含了 字符串中的每个 `Unicode` 符号的数字编码点 的数组。由于 `JavaScript` 在内部使用 `UCS-2` ，这个函数会将一对代理部分（surrogate halves）（UCS-2暴露的单独字符）转换为一个单独的编码点 来匹配UTF-16。

```
punycode.ucs2.decode('abc'); // [0x61, 0x62, 0x63]
// surrogate pair for U+1D306 tetragram for centre:
punycode.ucs2.decode('\uD834\uDF06'); // [0x1D306]
```

punycode.ucs2.encode(codePoints)

基于数字编码点的数组，创建一个字符串。

```
punycode.ucs2.encode([0x61, 0x62, 0x63]); // 'abc'
punycode.ucs2.encode([0x1D306]); // '\uD834\uDF06'
```

punycode.version

一个代表了当前 `Punycode.js` 版本号的数字。

Query String

稳定度: 2 - 稳定

这个模块提供了处理 查询字符串 的工具。它提供了以下方法：

querystring.stringify(obj[, sep][, eq][, options])

序列化一个对象为一个查询字符串。可以可选地覆盖默认的分隔符（ `'&'` ）和赋值符号（ `'='` ）。

`options` 对象可以包含 `encodeURIComponent` 属性（默认为 `querystring.escape` ），它被用来在需要时，将字符串编码为非utf-8编码。

例子：

```
querystring.stringify({ foo: 'bar', baz: ['qux', 'quux'], corge: '' })
// returns
'foo=bar&baz=qux&baz=quux&corge='

querystring.stringify({foo: 'bar', baz: 'qux'}, ';', ':')
// returns
'foo:bar;baz:qux'

// Suppose gbkEncodeURIComponent function already exists,
// it can encode string with `gbk` encoding
querystring.stringify({ w: '中文', foo: 'bar' }, null, null,
  { encodeURIComponent: gbkEncodeURIComponent })
// returns
'w=%D6%D0%CE%C4&foo=bar'
```

querystring.parse(str[, sep][, eq][, options])

反序列化一个查询字符串为一个对象。可以可选地覆盖默认的分隔符（ `'&'` ）和赋值符号（ `'='` ）。

`options` 可以包含 `maxKeys` 属性（默认为 `1000` ）。它被用来限制被处理的键。将其设置为 `0` 会移除限制。

`options` 可以包含 `decodeURIComponent` 属性（默认为 `querystring.unescape` ），它被用来在需要时，解码非uft8编码字符串。

例子：

```
querystring.parse('foo=bar&baz=qux&baz=quux&corge')
// returns
{ foo: 'bar', baz: ['qux', 'quux'], corge: '' }

// Suppose gbkDecodeURIComponent function already exists,
// it can decode `gbk` encoding string
querystring.parse('w=%D6%D0%CE%C4&foo=bar', null, null,
  { decodeURIComponent: gbkDecodeURIComponent })
// returns
{ w: '中文', foo: 'bar' }
```

querystring.escape

`querystring.stringify` 使用的转义函数，在需要时可以被覆盖。

querystring.unescape

`querystring.parse` 使用的反转义函数，在需要时可以被覆盖。

首先它会尝试使用 `decodeURIComponent` ，但是如果失败了，它就转而使用一个不会在畸形URL上抛出错误的更安全的等价方法。

Readline

稳定度: 2 - 稳定

通过 `require('readline')` 来使用这个模块。 `Readline` 允许逐行读取一个流（如 `process.stdin` ）。

注意，一旦你执行了这个模块，你的 `node.js` 程序在你关闭此接口之前，将不会退出。以下是如何让你的程序优雅的退出的例子：

```
var readline = require('readline');

var rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});

rl.question("What do you think of node.js? ", function(answer) {
  // TODO: Log the answer in a database
  console.log("Thank you for your valuable feedback:", answer);

  rl.close();
});
```

readline.createInterface(options)

创建一个 `readline` 接口实例。接受一个 `options` 对象，接受以下值：

- input - 需要监听的可读流（必选）。
- output - 将逐行读取的数据写入的流（可选）。
- completer - 用于Tab自动补全的可选函数。参阅下文的使用例子。
- terminal - 如果 `input` 和 `output` 流需要被像一个TTY一样对待，并且被经由ANSI/VT100转义代码写入，就传递 `true` 。默认为在实例化时，检查出的 `ouput` 流的 `isTTY` 值。

- historySize - 保留的历史记录行的最大数量。默认为 `30` 。

`completer` 函数被给予了一个用户输入的当前行，并且支持返回一个含有两个元素的数组：

1. 一个匹配当前输入补全的数组。
2. 一个被用于匹配的子字符串。

最终形式如： `[[substr1, substr2, ...], originalsubstring]` 。

例子：

```
function completer(line) {
  var completions = '.help .error .exit .quit .q'.split(' ')
  var hits = completions.filter(function(c) { return c.indexOf(l
ine) == 0 })
  // show all completions if none found
  return [hits.length ? hits : completions, line]
}
```

`completer` 同样也可以以同步的方式运行，如果它接受两个参数：

```
function completer(linePartial, callback) {
  callback(null, [['123'], linePartial]);
}
```

`createInterface` 通常与 `process.stdin` 和 `process.stdout` 搭配，用来接受用户输入：

```
var readline = require('readline');
var rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});
```

一旦你有了一个 `readline` 接口，你通常要监听一个 `line` 事件。

如果这个实例中， `terminal` 为 `true` 。那么 `output` 流将会得到最好的兼容性，如果它定义了 `output.columns` 属性，并且在 `output` 的 `columns` 变化时（当它是一个TTY时， `process.stdout` 会自动这么做），触发了一个 `resize` 事件。

Class: Interface

一个代表了有 `input` 和 `output` 流的 `readline` 接口。

rl.setPrompt(prompt)

设置提示符，例如当你在命令行运行 `iojs` 命令时，你看到了 `>` ，这就是 `node.js` 的提示符。

rl.prompt([preserveCursor])

为用户的输入准备好 `readline` ，在新的一行放置当前的 `setPrompt` 选项，给予用户一个新的用于输入的地方。设置 `preserveCursor` 为 `true` ，来防止光标位置被重置为 `0` 。

仍会重置被 `createInterface` 使用的 `input` 流，如果它被暂停。

如果当调用 `createInterface` 时 `output` 被设置为 `null` 或 `undefined` ，提示符将不会被写入。

rl.question(query, callback)

带着 `query` 来预先放置提示符，并且在用户应答时执行回调函数。给用户展示 `query` ，然后在用户输入了应答后调用 `callback` 。

仍会重置被 `createInterface` 使用的 `input` 流，如果它被暂停。

如果当调用 `createInterface` 时 `output` 被设置为 `null` 或 `undefined` ，什么都不会被展示。

例子：

```
interface.question('What is your favorite food?', function(answer) {
  console.log('Oh, so your favorite food is ' + answer);
});
```

rl.pause()

暂停 `readline` 的 `input` 流，允许它在晚些需要时恢复。

注意，带着事件的流不会立刻被暂停。在调用了 `pause` 后，许多事件可能被触发，包括 `line` 事件。

rl.resume()

恢复 `readline` 的 `input` 流。

rl.close()

关闭实例接口，放弃对 `input` 和 `output` 流的控制。 `close` 事件也会被触发。

rl.write(data[, key])

向 `output` 流写入数据，除非当调用 `createInterface` 时 `output` 被设置为 `null` 或 `undefined` 。 `key` 是一个代表了键序列的对象；在当终端为TTY时可用。

如果 `input` 流被暂停，它也会被恢复。

例子：

```
rl.write('Delete me!');
// Simulate ctrl+u to delete the line written previously
rl.write(null, {ctrl: true, name: 'u'});
```

Events

Event: 'line'

- function (line) {}

当 `input` 流收到一个 `\n` 时触发，通常在用户敲下回车时触发。这是一个监听用户输入的好钩子。

例子：

```
rl.on('line', function (cmd) {
  console.log('You just typed: '+cmd);
});
```

Event: 'pause'

- function () {}

当 `input` 流被暂停时触发。

也会在 `input` 没有被暂停并且收到一个 `SIGCONT` 事件时触发（参阅 `SIGTSTP` 事件和 `SIGCONT` 事件）。

例子：

```
rl.on('pause', function() {
  console.log('Readline paused.');
});
```

Event: 'resume'

- function () {}

当 `input` 流被恢复时触发。

例子：

```
rl.on('resume', function() {
  console.log('Readline resumed.');
});
```

Event: 'close'

- function () {}

当 `close()` 被调用时触发。

也会在 `input` 流收到它的 `end` 事件时触发。当这个事件触发时，接口实例需要考虑“被结束”。例如，当 `input` 流接收到 `^D` （也被认作 `EOT` ）。

这个事件也会在如果当前没有 `SIGINT` 事件监听器，且 `input` 流接收到 `^C` （也被认作 `SIGINT` ）时触发。

Event: 'SIGINT'

- function () {}

当 `input` 流接收到 `^C` （也被认作 `SIGINT` ）时触发。如果当前没有 `SIGINT` 事件的监听器， `pause` 事件将会被触发。

例子：

```
rl.on('SIGINT', function() {
  rl.question('Are you sure you want to exit?', function(answer)
{
    if (answer.match(/^y(es)?$/i)) rl.pause();
  });
});
```

Event: 'SIGTSTP'

- function () {}

在Windows平台下不能使用。

当 `input` 流接收到一个 `^Z` （也被认作 `SIGTSTP` ）时触发。如果当前没有 `SIGTSTP` 事件的监听器，这个程序将会被送至后台运行。

当程序使用 `fg` 恢复， `pause` 和 `SIGCONT` 事件都会被触发。你可以选择其中的一个来恢复流。

如果流在程序被送至后台前就被暂停， `pause` 和 `SIGCONT` 事件将不会触发。

例子：

```
rl.on('SIGTSTP', function() {
  // This will override SIGTSTP and prevent the program from goi
ng to the
  // background.
  console.log('Caught SIGTSTP.');
});
```

Event: 'SIGCONT'

- function () {}

在Windows平台下不能使用。

当 `input` 流被 `^Z` （也被认作 `SIGTSTP` ）送至后台时触发，然后使用 `fg(1)` 继续执行。这个事件仅在程序被送至后台前流没有被暂停时触发。

例子：

```
rl.on('SIGCONT', function() {
  // `prompt` will automatically resume the stream
  rl.prompt();
});
```

Example: Tiny CLI

下面是一个使用以上方法来创建一个迷你的控制台接口的例子：

```
var readline = require('readline'),
    rl = readline.createInterface(process.stdin, process.stdout)
;

rl.setPrompt('OHAI> ');
rl.prompt();

rl.on('line', function(line) {
  switch(line.trim()) {
    case 'hello':
      console.log('world!');
      break;
    default:
      console.log('Say what? I might have heard `' + line.trim()
 + '`');
      break;
  }
  rl.prompt();
}).on('close', function() {
  console.log('Have a great day!');
  process.exit(0);
});
```

readline.cursorTo(stream, x, y)

在给定的TTY流中，将光标移动到指定位置。

readline.moveCursor(stream, dx, dy)

在给定的TTY流中，相对于当前位置，将光标移动到指定位置。

readline.clearLine(stream, dir)

用指定的方式，在给定的TTY流中，清除当前的行。 `dir` 可以是以下值之一：

- -1 - 从光标的左边
- 1 - 从光标的右边
- 0 - 整行

readline.clearScreenDown(stream)

从当前的光标位置，清除屏幕。

REPL

稳定度: 2 - 稳定

一个 读取-执行-打印-循环（REPL）可以用于单独的程序，也能很容易的被集成在其他程序中。`REPL` 提供了一种交互着运行 `JavaScript` 然后查看结果的方式。它可以被用来调试，测试或只是尝试一些东西。

在命令行中不带任何参数直接执行 `iojs` ，你会进入REPL界面。它有一个极简的emacs行编辑器。

```
mjr:~$ iojs
Type '.help' for options.
> a = [ 1, 2, 3];
[ 1, 2, 3 ]
> a.forEach(function (v) {
...   console.log(v);
...   });
1
2
3
```

要使用高级的行编辑器的话，带着环境变量 `NODE_NO_READLINE=1` 启动 `node.js` 。它将会在允许你使用 `rlwrap` 的终端设置中，启动一个主要的调试 `REPL` （main and debugger REPL）。

例如，你可以把以下内容加入 `bashrc` 文件：

```
alias iojs="env NODE_NO_READLINE=1 rlwrap iojs"
```

内置的 `REPL` （通过运行 `iojs` 或 `iojs -i` 启动）可以被以下环境变量所控制：

- NODE_REPL_HISTORY_FILE - 如果指定，那必须是一个用户可读也可写的文件路径。当给定了一个可用的路径，将启用持久化的历史记录支持：`REPL` 历史记录将会跨 `iojs``REPL` 会话持久化。
- NODE_REPL_HISTORY_SIZE - 默认为 `1000` 。

与 `NODE_REPL_HISTORY_FILE` 结合，控制需要持久化的历史记录数量。必须为正数。
- NODE_REPL_MODE - 可以是 `sloppy` ， `strict` 或 `magic` 中的一个。默认为 `magic` ，会自动在严格模式中执行 `"strict mode only"` 声明。

repl.start(options)

返回并启动一个 `REPLServer` 实例，继承于 `[Readline Interface][]` 。接受一个包含以下值得 `options` 对象：

- prompt - 所有 `I/O` 的提示符。默认为 `>` 。
- input - 监听的可读流。默认为 `process.stdin` 。
- output - 输出数据的可写流。默认为 `process.stdout` 。
- terminal - 如果流需要被像TTY对待，并且有 `ANSI/VT100` 转义代码写入，设置其为 `true` 。默认为在实例化时检查到的 `output` 流的 `isTTY` 属性。
- eval - 被用来执行每一行的函数。默认为被异步包装过的 `eval()` 。参阅下文的自定义 `eval` 的例子。
- useColors - 一个表明了是否 `writer` 函数需要输出颜色的布尔值。如果设置了不同的 `writer` 函数，那么它什么都不会做。默认为 `REPL` 的终端值。
- useGlobal - 若设置为 `true` ，那么 `REPL` 将使用全局对象，而不是运行每一个脚本在不同上下文中。默认为 `false` 。
- ignoreUndefined - 若设置为 `true` ，那么如果返回值是 `undefined` ， `REPL` 将不会输出它。默认为 `false` 。
- writer - 当每一个命令被执行完毕时，都会调用这个函数，它返回了展示的格式（包括颜色）。默认为 `util.inspect` 。
- **replMode** - 控制是否 `REPL` 运行所有的模式在严格模式，默认模式，或混合模式（ `"magic"` 模式）。接受以下值：
    - repl.REPL_MODE_SLOPPY - 在混杂模式下运行命令。
    - repl.REPL_MODE_STRICT - 在严格模式下运行命令。这与在每个命令前添加 `'use strict'` 语句相等。
    - repl.REPL_MODE_MAGIC - 试图在默认模式中运行命令，如果失败了，会重新尝试使用严格模式。

你可以使用你自己的 `eval` 函数，如果它包含以下签名：

```
function eval(cmd, context, filename, callback) {
  callback(null, result);
}
```

在用tab补全时 - `eval` 将会带着一个作为输入字符串的 `.scope` 调用。它被期望返回一个 `scope` 名字数组，被用来自动补全。

多个 `REPL` 可以运行相同的 `node.js` 实例。共享同一个全局对象，但是各自的I/O独立。

下面是在 `stdin` ，Unix `socket` 和 TCP `socket` 上启动一个 `REPL` 的例子：

```
var net = require("net"),
    repl = require("repl");

connections = 0;

repl.start({
  prompt: "node.js via stdin> ",
  input: process.stdin,
  output: process.stdout
});

net.createServer(function (socket) {
  connections += 1;
  repl.start({
    prompt: "node.js via Unix socket> ",
    input: socket,
    output: socket
  }).on('exit', function() {
    socket.end();
  })
}).listen("/tmp/iojs-repl-sock");

net.createServer(function (socket) {
  connections += 1;
  repl.start({
    prompt: "node.js via TCP socket> ",
    input: socket,
    output: socket
  }).on('exit', function() {
    socket.end();
  });
}).listen(5001);
```

在命令行中运行这个程序会在 `stdin` 上启动一个 `REPL` 。另外的 `REPL` 客户端将会通过Unix `socket` 或TCP `socket` 连接。 `telnet` 在连接TCP `socket` 时非常有用， `socat` 在连接Unix `socket` 和TCP `socket` 时都非常有用。

通过从基于Unix `socket` 的服务器启动 `REPL` ，你可以不用重启，而连接到一个长久执行的（long-running） `node.js` 进程。

一个通过 `net.Server` 和 `net.Socket` 实例运行"全特性"（终端） `REPL` 的例子，参阅 `https://gist.github.com/2209310` 。

一个通过 `curl(1)` 运行 `REPL` 的例子，参阅 `https://gist.github.com/2053342` 。

Event: 'exit'

- function () {}

当用户通过任意一种已定义的方式退出 `REPL` 时触发。具体地说，在 `REPL` 中键入 `.exit` ，两次按下 `Ctrl+C` 来发送 `SIGINT` 信号，按下 `Ctrl+D` 来发送结束信号。

例子：

```
r.on('exit', function () {
  console.log('Got "exit" event from repl!');
  process.exit();
});
```

Event: 'reset'

- function (context) {}

当 `REPL` 内容被重置时触发。当你键入 `.clear` 时发生。如果你以 `{ useGlobal: true }` 启动 `REPL` ，那么这个事件将永远不会触发。

例子：

```
// Extend the initial repl context.
r = repl.start({ options ... });
someExtension.extend(r.context);

// When a new context is created extend it as well.
r.on('reset', function (context) {
  console.log('repl has a new context');
  someExtension.extend(context);
});
```

REPL 特性

在 `REPL` 内，按下 `Control+D` 将会退出。多行表达式可以被输入。Tab补全同时支持全局和本地变量。

核心模块将会被按需载入环境。例如，调用 `fs` ，将会从 `global.fs` 获取，作为 `require()``fs` 模块的替代。

特殊的变量 `_` （下划线）包含了上一个表达式的结果。

```
> [ "a", "b", "c" ]
[ 'a', 'b', 'c' ]
> _.length
3
> _ += 1
4
```

`REPL` 可以访问全局作用域里的任何变量。你可以通过将变量赋值给一个关联了所有 `REPLServer` 的 `context` 对象来暴露一个对象给 `REPL` 。例子：

```
// repl_test.js
var repl = require("repl"),
    msg = "message";

repl.start("> ").context.m = msg;
```

`context` 对象里的对象会表现得像 `REPL` 的本地变量：

```
mjr:~$ iojs repl_test.js
> m
'message'
```

以下是一些特殊的 `REPL` 命令：

- .break - 当你输入一个多行表达式时，有时你走神了，或有时你不关心如何完成它了。 `.break` 将会让你重新来过。
- .clear - 重置 `context` 对象为一个空对象并且清除所有多行表达式。
- .exit - 关闭I/O流，意味着会导致 `REPL` 退出。

- .help - 展示特殊命令列表。
- **.save** 将当前的 `REPL` 会话保存入一个文件。
    - `.save ./file/to/save.js`
- **.load** - 从一个文件中加载 `REPL` 会话。
    - `.load ./file/to/load.js`

这些组合键在 `REPL` 中有以下影响：

- ctrl + C - 与 `.break` 关键字相似。终止当前命令。在一个空行上连按两次会强制退出。
- ctrl + D - 与 `.exit` 关键字相似。
- tab - 展示所有的全局和本地变量。

Stream

稳定度: 2 - 稳定

流是一个被 `node.js` 内部的许多对象所实现的抽象接口。例如一个发往HTTP服务器的请求是一个留， `stdout` 也是一个流。流可以是可读的，可写的或双向的。所有的流都是 `EventEmitter` 实例。

你可以通过 `require('stream')` 来取货 `Stream` 的基类。其中包括了 `Readable` 流， `Writable` 流， `Duplex` 流和 `Transform` 流的基类。

此文档分为三个章节。第一章节解释了在你的编程中使用流时需要的API。如果你不需要实现你自己的流式API，你可以在这里停止。

第二章节解释了你在构建你自己的流时需要的API，这些API是为了方便你这么做而设计的。

第三章节深入讲述了流的工作机制，包括一些内部的机制和函数，你不应该去改动它们除非你知道你在做什么。

面向流消费者的API

流可以是可读的，可写的，或双工的。

所有的流都是 `EventEmitters` 。但是它们也各自有一些独特的方法和属性，这取决于它们是可读流，可写流或双工流。

如果一个流同时是可读的和可写的，那么表示它实现了以下所有的方法和事件。所以，这些API同时也涵盖 `Duplex` 或 `Transform` 流，即使它们的实现可能有些不同。

在你程序中，为了消费流而去实现流接口不是必须的。如果你确实正在你的程序中实现流接口，请参考下一章节 `面向流实现者的API` 。

几乎所有 `node.js` 程序，不论多简单，都使用了流。下面是一个在 `node.js` 是使用流的例子：

```
var http = require('http');
```

```
var server = http.createServer(function (req, res) {
  // req is an http.IncomingMessage, which is a Readable Stream
  // res is an http.ServerResponse, which is a Writable Stream

  var body = '';
  // we want to get the data as utf8 strings
  // If you don't set an encoding, then you'll get Buffer objects

  req.setEncoding('utf8');

  // Readable streams emit 'data' events once a listener is added

  req.on('data', function (chunk) {
    body += chunk;
  });

  // the end event tells you that you have entire body
  req.on('end', function () {
    try {
      var data = JSON.parse(body);
    } catch (er) {
      // uh oh!  bad json!
      res.statusCode = 400;
      return res.end('error: ' + er.message);
    }

    // write back something interesting to the user:
    res.write(typeof data);
    res.end();
  });
});

server.listen(1337);

// $ curl localhost:1337 -d '{}'
// object
// $ curl localhost:1337 -d '"foo"'
// string
// $ curl localhost:1337 -d 'not json'
```

```
// error: Unexpected token o
```

Class: stream.Readable

可读流接口是一个你可以从之读取数据的数据源的抽象。换句话说，数据从可读流而来。

除非你指示已经准备好接受数据，否则可读流不会开始发生数据。

可读流有两个“模式”：流动模式和暂停模式。当在流动模式时，数据由底层系统读出，并且会尽快地提供给你的程序。当在暂停模式时，你必须调用 `stream.read()` 方法来获取数据块。流默认是暂停模式。

注意：如果 `data` 事件没有被绑定监听器，并且没有导流（pipe）目标，并且流被切换到了流动模式，那么数据将会被丢失。

你可以通过下面任意一个做法切换到流动模式：

- 添加一个 `data` 事件的监听器来监听数据。
- 调用 `resume()` 方法来明确开启流动模式。
- 调用 `pipe()` 方法将数据导入一个可写流。

你可以同意下面任意一种方法切换回暂停模式：

- 如果没有导流（pipe）目标，调用 `pause()` 方法。
- 如果有导流（pipe）目标，移除所有的 `data` 事件监听器，并且通过 `unpipe()` 方法移除所有导流目标。

注意，由于为了向后兼任的原因，移除 `data` 事件的监听器将不会自动暂停流。同样的，如果有导流目标，调用 `pause()` 方法将不会保证目标流排空并请求更多数据时保持暂停。

一些内置的可读流例子：

- 客户端的HTTP请求
- 服务端的HTTP响应
- 文件系统读取流
- `zlib` 流

- `crypto` 流
- tcp sockets
- 子进程的stdout和stderr
- `process.stdin`

Event: 'readable'

当一个数据块能可以从流中被读出时，会触发一个 `readable` 事件。

某些情况下，监听一个 `readable` 事件会导致一些将要被读出的数据从底层系统进入内部缓冲，如果它没有准备好。

```
var readable = getReadableStreamSomehow();
readable.on('readable', function() {
  // there is some data to read now
});
```

当内部缓冲被排空时，一旦有更多数据， `readable` 事件会再次触发。

Event: 'data'

- chunk Buffer | String 数据块

为一个没有被暂停的流添加一个 `data` 事件的监听器会使其切换到流动模式。之后数据会被尽快得传递给用户。

如果你只是想尽快得从流中取得所有数据，这是最好的方式。

```
var readable = getReadableStreamSomehow();
readable.on('data', function(chunk) {
  console.log('got %d bytes of data', chunk.length);
});
```

Event: 'end'

当没有更多可读的数据时这个事件会被触发。

注意，除非数据被完全消费， `end` 事件才会触发。这可以通过切换到流动模式，或重复调用 `read()` 方法。

```
var readable = getReadableStreamSomehow();
readable.on('data', function(chunk) {
  console.log('got %d bytes of data', chunk.length);
});
readable.on('end', function() {
  console.log('there will be no more data.');
});
```

Event: 'close'

当底层资源（如源头的文件描述符）被关闭时触发。不是所有的流都会触发这个事件。

Event: 'error'

- Error Object

当接受数据时有错误发生，会触发此事件。

readable.read([size])

- size Number 可选，指定读取数据的数量
- Return String | Buffer | null

`read()` 方法从内部缓冲中取出数据并返回它。如果没有可用数据，那么将返回 `null` 。

如果你传递了一个 `size` 参数，那么它将返回指定字节的数据。如果 `size` 参数的字节数不可用，那么将返回 `null` 。

如果你不指定 `size` 参数，那么将会返回内部缓冲中的所有数据。

这个方法只能在暂定模式中被调用。在流动模式下，这个方法会被自动地重复调用，知道内部缓冲被排空。

```
var readable = getReadableStreamSomehow();
readable.on('readable', function() {
  var chunk;
  while (null !== (chunk = readable.read())) {
    console.log('got %d bytes of data', chunk.length);
  }
});
```

如果这个方法返回一个数据块，那么它也会触发 `data` 事件。

readable.setEncoding(encoding)

- encoding String 使用的编码
- Return: this

调用这个函数会导致流返回指定编码的字符串而不是 `Buffer` 对象。例如，如果你调用 `readable.setEncoding('utf8')` ，那么输出的数据将被解释为UTF-8数据，并且作为字符串返回。如果你调用了 `readable.setEncoding('hex')` ，那么数据将被使用十六进制字符串的格式编码。

该方法可以正确地处理多字节字符。如果你只是简单地直接取出缓冲并且对它们调用 `buf.toString(encoding)` ，将会导致错位。如果你想使用字符串读取数据，请使用这个方法。

```
var readable = getReadableStreamSomehow();
readable.setEncoding('utf8');
readable.on('data', function(chunk) {
  assert.equal(typeof chunk, 'string');
  console.log('got %d characters of string data', chunk.length);
});
```

readable.resume()

- Return: this

这个方法将会让可读流继续触发 `data` 事件。

这个方法将会使流切换至流动模式。如果你不想消费流中的数据，但你想监听它的 `end` 事件，你可以通过调用 `readable.resume()` 来打开数据流。

```
var readable = getReadableStreamSomehow();
readable.resume();
readable.on('end', function() {
  console.log('got to the end, but did not read anything');
});
```

readable.pause()

- Return: this

这个方法会使一个处于流动模式的流停止触发 `data` 事件，并切换至暂停模式。所有可用的数据将仍然存在于内部缓冲中。

```
var readable = getReadableStreamSomehow();
readable.on('data', function(chunk) {
  console.log('got %d bytes of data', chunk.length);
  readable.pause();
  console.log('there will be no more data for 1 second');
  setTimeout(function() {
    console.log('now data will start flowing again');
    readable.resume();
  }, 1000);
});
```

readable.isPaused()

- Return: Boolean

这个方法会返回流是否被客户端代码所暂停（调用 `readable.pause()` ，并且没有在之后调用 `readable.resume()` ）。

```
var readable = new stream.Readable

readable.isPaused() // === false
readable.pause()
readable.isPaused() // === true
readable.resume()
readable.isPaused() // === false
```

readable.pipe(destination[, options])

- destination Writable Stream 写入数据的目标
- **options Object**
    - end Boolean 当读取者结束时结束写入者。默认为 `true` 。

这个方法会取出可读流中所有的数据，并且将之写入指定的目标。这个方法会自动调节流量，所以当快速读取可读流时目标不会溢出。

可以将数据安全地导流至多个目标。

```
var readable = getReadableStreamSomehow();
var writable = fs.createWriteStream('file.txt');
// All the data from readable goes into 'file.txt'
readable.pipe(writable);
```

这个函数返回目标流，所以你可以链式调用 `pipe()` ：

```
var r = fs.createReadStream('file.txt');
var z = zlib.createGzip();
var w = fs.createWriteStream('file.txt.gz');
r.pipe(z).pipe(w);
```

例子，模仿UNIX的 `cat` 命令：

```
process.stdin.pipe(process.stdout);
```

默认情况下，当源流触发 `end` 事件时，目标流会被调用 `end()` 方法，然后目标就不再是可写的了。将传递 `{ end: false }` 作为 `options` 参数，将保持目标流开启。

例子，保持被写入的流开启，所以“Goodbye”可以在末端被写入：

```
reader.pipe(writer, { end: false });
reader.on('end', function() {
  writer.end('Goodbye\n');
});
```

注意，不论指定任何 `options` 参数， `process.stderr` 和 `process.stdout` 在程序退出前永远不会被关闭。

readable.unpipe([destination])

- destination Writable Stream 可选，指定解除导流的流

这方法会移除之前调用 `pipe()` 方法所设置的钩子。

如果没有指定目标，那么所有的导流都会被移除。

如果指定了目标，但是并没有为目标设置导流，那么什么都不会发生。

```
var readable = getReadableStreamSomehow();
var writable = fs.createWriteStream('file.txt');
// All the data from readable goes into 'file.txt',
// but only for the first second
readable.pipe(writable);
setTimeout(function() {
  console.log('stop writing to file.txt');
  readable.unpipe(writable);
  console.log('manually close the file stream');
  writable.end();
}, 1000);
```

readable.unshift(chunk)

- chunk Buffer | String 要插回读取队列开头的数据块。

该方法在许多场景中都很有用，比如一个流正在被一个解析器消费，解析器可能需要将某些刚拉取出的数据“逆消费”回来源，以便流能将它传递给其它消费者。

如果你发现你必须经常在你的程序中调用 `stream.unshift(chunk)` ，你应该考虑实现一个 `Transform` 流（参阅下文的面向流实现者的API）。

```
// Pull off a header delimited by \n\n
// use unshift() if we get too much
// Call the callback with (error, header, stream)
var StringDecoder = require('string_decoder').StringDecoder;
function parseHeader(stream, callback) {
  stream.on('error', callback);
  stream.on('readable', onReadable);
  var decoder = new StringDecoder('utf8');
  var header = '';
  function onReadable() {
    var chunk;
    while (null !== (chunk = stream.read())) {
      var str = decoder.write(chunk);
      if (str.match(/\n\n/)) {
        // found the header boundary
        var split = str.split(/\n\n/);
        header += split.shift();
        var remaining = split.join('\n\n');
        var buf = new Buffer(remaining, 'utf8');
        if (buf.length)
          stream.unshift(buf);
        stream.removeListener('error', callback);
        stream.removeListener('readable', onReadable);
        // now the body of the message can be read from the stre
am.
        callback(null, header, stream);
      } else {
        // still reading the header.
        header += str;
      }
    }
  }
}
```

readable.wrap(stream)

- stream Stream 一个“旧式”可读流

`Node.js` v0.10 以及之前版本的流没有完全包含如今的所有的流API（更多的信息请参阅下文的“兼容性”）。

如果你正在使用一个老旧的 `node.js` 库，它触发 `data` 时间并且有一个仅作查询用途的 `pause()` 方法，那么你可以调用 `wrap()` 方法来创建一个使用“旧式”流作为数据源的可读流。

你几乎不会用到这个函数，它的存在仅是为了老旧的 `node.js` 程序和库交互。

例子：

```
var OldReader = require('./old-api-module.js').OldReader;
var oreader = new OldReader;
var Readable = require('stream').Readable;
var myReader = new Readable().wrap(oreader);

myReader.on('readable', function() {
  myReader.read(); // etc.
});
```

Class: stream.Writable

可写流接口是一个你可以向其写入数据的目标的抽象。

一些内部的可写流例子：

- 客户端的http请求
- 服务端的http响应
- 文件系统写入流
- `zlib` 流
- `crypto` 流
- tcp `socket`
- 子进程 `stdin`
- `process.stdout` ， `process.stderr`

writable.write(chunk[, encoding][, callback])

- chunk String | Buffer 要写入的数据
- encoding String 编码，如果数据块是字符串
- callback Function 当数据块写入完毕后调用的回调函数
- Returns: Boolean 如果被全部处理则返回 `true`

该方法向底层系统写入数据，并且当数据被全部处理后调用指定的回调函数。

返回值指示了你是否可以立刻写入数据。如果数据需要被内部缓冲，会返回 `false` 。否则返回 `true` 。

返回值经供参考。即使返回 `false` ，你仍可以继续写入数据。但是，写入的数据将会被缓冲在内存里，所以最好不要这样做。应该在写入更多数据前等待 `drain` 事件。

Event: 'drain'

如果一个 `writable.write(chunk)` 调用返回了 `false` ，那么 `drain` 事件会指示出可以继续向流写入数据的时机。

```
// Write the data to the supplied writable stream 1MM times.
// Be attentive to back-pressure.
function writeOneMillionTimes(writer, data, encoding, callback)
{
  var i = 1000000;
  write();
  function write() {
    var ok = true;
    do {
      i -= 1;
      if (i === 0) {
        // last time!
        writer.write(data, encoding, callback);
      } else {
        // see if we should continue, or wait
        // don't pass the callback, because we're not done yet.
        ok = writer.write(data, encoding);
      }
    } while (i > 0 && ok);
    if (i > 0) {
      // had to stop early!
      // write some more once it drains
      writer.once('drain', write);
    }
  }
}
```

writable.cork()

强制滞留所有写入。

滞留的数据会在调用 `.uncork()` 或 `.end()` 方法后被写入。

writable.uncork()

写入在调用 `.cork()` 方法所有被滞留的数据。

writable.setDefaultEncoding(encoding)

- encoding String 新的默认编码

设置一个可写流的默认编码。

writable.end([chunk][, encoding][, callback])

- chunk String | Buffer 可选，写入的数据
- encoding String 编码，如果数据块是字符串
- callback Function 可选，回调函数

当没有更多可写的数据时，调用这个方法。如果指定了回调函数，那么会被添加为 `finish` 事件的监听器。

在调用了 `end()` 后调用 `write()` 会导致一个错误。

```
// write 'hello, ' and then end with 'world!'
var file = fs.createWriteStream('example.txt');
file.write('hello, ');
file.end('world!');
// writing more now is not allowed!
```

Event: 'finish'

当调用了 `end()` 方法，并且所有的数据都被写入了底层系统，这个事件会被触发。

```
var writer = getWritableStreamSomehow();
for (var i = 0; i < 100; i ++) {
  writer.write('hello, #' + i + '!\n');
}
writer.end('this is the end\n');
writer.on('finish', function() {
  console.error('all writes are now complete.');
});
```

Event: 'pipe'

- src Readable Stream 对这个可写流进行导流的源可读流

这个事件将会在可读流被一个可写流使用 `pipe()` 方法进行导流时触发。

```
var writer = getWritableStreamSomehow();
var reader = getReadableStreamSomehow();
writer.on('pipe', function(src) {
  console.error('something is piping into the writer');
  assert.equal(src, reader);
});
reader.pipe(writer);
```

Event: 'unpipe'

- src Readable Stream 对这个可写流停止导流的源可读流

当可读流对其调用 `unpipe()` 方法，在源可读流的目标集合中删除这个可写流，这个事件将会触发。

```
var writer = getWritableStreamSomehow();
var reader = getReadableStreamSomehow();
writer.on('unpipe', function(src) {
  console.error('something has stopped piping into the writer');
  assert.equal(src, reader);
});
reader.pipe(writer);
reader.unpipe(writer);
```

Event: 'error'

- Error object

在写入数据或导流发生错误时触发。

Class: stream.Duplex

双工是同时实现了可读流与可写流的借口。它的用处请参阅下文。

内部双工流的例子：

- tcp `socket`

- `zlib` 流
- `crypto` 流

Class: stream.Transform

转换流是一种输出由输入计算所得的栓共流。它们同时集成了可读流与可写流的借口。它们的用处请参阅下文。

内部转换流的例子：

- `zlib` 流
- `crypto` 流

面向流实现者的API

实现所有种类的流的模式都是一样的：

1. 为你的子类继承合适的父类（ `util.inherits` 非常合适于做这个）。
2. 为了保证内部机制被正确初始化，在你的构造函数中调用合适的父类构造函数。
3. 实现一个或多个特定的方法，参阅下文。

被扩展的类和要实现的方法取决于你要编写的流类的类型：

用途	类	需要实现的方法
只读	Readable	_read
只写	Writable	_write, _writev
可读以及可写	Duplex	_read, _write, _writev
操作被写入数据，然后读出结果	Transform	_transform, _flush

在你的实现代码中，非常重要的一点是永远不要调用上文的面向流消费者的API。否则，你在程序中消费你的流接口时可能有潜在的副作用。

Class: stream.Readable

`stream.Readable` 是一个被设计为需要实现底层的 `_read(size)` 方法的抽象类。

请参阅上文的面向流消费者的API来了解如何在程序中消费流。以下解释了如果在你的程序中实现可读流。

例子：一个计数流

这是一个可读流的基础例子。它从1到1，000，000递增数字，然后结束。

```
var Readable = require('stream').Readable;
var util = require('util');
util.inherits(Counter, Readable);

function Counter(opt) {
  Readable.call(this, opt);
  this._max = 1000000;
  this._index = 1;
}

Counter.prototype._read = function() {
  var i = this._index++;
  if (i > this._max)
    this.push(null);
  else {
    var str = '' + i;
    var buf = new Buffer(str, 'ascii');
    this.push(buf);
  }
};
```

例子：简单协议 v1 （次优）

这类似于上文中提到的 `parseHeader` 函数，但是使用一个自定义流实现。另外，注意这个实现不将流入的数据转换为字符串。

更好地实现是作为一个转换流实现，请参阅下文更好地实现。

```
// A parser for a simple data protocol.
// The "header" is a JSON object, followed by 2 \n characters, a
nd
// then a message body.
//
```

```
// NOTE: This can be done more simply as a Transform stream!
// Using Readable directly for this is sub-optimal.  See the
// alternative example below under the Transform section.

var Readable = require('stream').Readable;
var util = require('util');

util.inherits(SimpleProtocol, Readable);

function SimpleProtocol(source, options) {
  if (!(this instanceof SimpleProtocol))
    return new SimpleProtocol(source, options);

  Readable.call(this, options);
  this._inBody = false;
  this._sawFirstCr = false;

  // source is a readable stream, such as a socket or file
  this._source = source;

  var self = this;
  source.on('end', function() {
    self.push(null);
  });

  // give it a kick whenever the source is readable
  // read(0) will not consume any bytes
  source.on('readable', function() {
    self.read(0);
  });

  this._rawHeader = [];
  this.header = null;
}

SimpleProtocol.prototype._read = function(n) {
  if (!this._inBody) {
    var chunk = this._source.read();

    // if the source doesn't have data, we don't have data yet.
```

```
    if (chunk === null)
      return this.push('');

    // check if the chunk has a \n\n
    var split = -1;
    for (var i = 0; i < chunk.length; i++) {
      if (chunk[i] === 10) { // '\n'
        if (this._sawFirstCr) {
          split = i;
          break;
        } else {
          this._sawFirstCr = true;
        }
      } else {
        this._sawFirstCr = false;
      }
    }

    if (split === -1) {
      // still waiting for the \n\n
      // stash the chunk, and try again.
      this._rawHeader.push(chunk);
      this.push('');
    } else {
      this._inBody = true;
      var h = chunk.slice(0, split);
      this._rawHeader.push(h);
      var header = Buffer.concat(this._rawHeader).toString();
      try {
        this.header = JSON.parse(header);
      } catch (er) {
        this.emit('error', new Error('invalid simple protocol da
ta'));
        return;
      }
      // now, because we got some extra data, unshift the rest
      // back into the read queue so that our consumer will see
it.
      var b = chunk.slice(split);
      this.unshift(b);
```

```
      // and let them know that we are done parsing the header.
      this.emit('header', this.header);
    }
  } else {
    // from there on, just provide the data to our consumer.
    // careful not to push(null), since that would indicate EOF.
    var chunk = this._source.read();
    if (chunk) this.push(chunk);
  }
};

// Usage:
// var parser = new SimpleProtocol(source);
// Now parser is a readable stream that will emit 'header'
// with the parsed header data.
```

new stream.Readable([options])

- **options Object**
  - highWaterMark Number 在停止从底层资源读取之前，在内部缓冲中存储的最大字节数。默认为16kb，对于 `objectMode` 则是16
  - encoding String 如果被指定，那么缓冲将被利用指定编码解码为字符串，默认为 `null`
  - objectMode Boolean 是否该流应该表现如一个对象的流。意思是说 `stream.read(n)` 返回一个单独的对象而不是一个大小为 `n` 的 `Buffer` ，默认为 `false`

在实现了 `Readable` 类的类中，请确保调用了 `Readable` 构造函数，这样缓冲设置才能被正确的初始化。

readable._read(size)

- size Number 异步读取数据的字节数

注意：实现这个函数，而不要直接调用这个函数。

这个函数不应该被直接调用。它应该被子类实现，并且仅被 `Readable` 类的内部方法调用。

所有的可读流都必须实现这个方法用来从底层资源中获取数据。

这个函数有一个下划线前缀，因为它对于类是内部的，并应该直接被用户的程序调用。你应在你的拓展类里覆盖这个方法。

当数据可用时，调用 `readable.push(chunk)` 方法将之推入读取队列。如果方法返回 `false` ，那么你应当停止读取。当 `_read` 方法再次被调用，你应当推入更多数据。

参数 `size` 仅作查询。“read”调用返回数据的实现可以通过这个参数来知道应当抓取多少数据；其余与之无关的实现，比如TCP或TLS，则可忽略这个参数，并在可用时返回数据。例如，没有必要“等到” `size` 个字节可用时才调用 `stream.push(chunk)` 。

readable.push(chunk[, encoding])

- chunk Buffer | null | String 被推入读取队列的数据块
- encoding String 字符串数据块的编码。必须是一个合法的 `Buffer` 编码，如'utf8'或'ascii'
- return Boolean 是否应该继续推入

注意：这个函数应该被 `Readable` 流的实现者调用，而不是消费者。

`_read()` 函数在至少调用一次 `push(chunk)` 方法前，不会被再次调用。

`Readable` 类通过在 `readable` 事件触发时，调用 `read()` 方法将数据推入 之后用于读出数据的读取队列 来工作。

`push()` 方法需要明确地向读取队列中插入数据。如果它的参数为 `null` ，那么它将发送一个数据结束信号（ `EOF` ）。

这个API被设计为尽可能的灵活。例如，你可能正在包装一个有 `pause/resume` 机制和一个数据回调函数的低级别源。那那些情况下，你可以通过以下方式包装这些低级别源：

```
// source is an object with readStop() and readStart() methods,
// and an `ondata` member that gets called when it has data, and
// an `onend` member that gets called when the data is over.

util.inherits(SourceWrapper, Readable);

function SourceWrapper(options) {
  Readable.call(this, options);

  this._source = getLowlevelSourceObject();
  var self = this;

  // Every time there's data, we push it into the internal buffe
r.
  this._source.ondata = function(chunk) {
    // if push() returns false, then we need to stop reading fro
m source
    if (!self.push(chunk))
      self._source.readStop();
  };

  // When the source ends, we push the EOF-signaling `null` chunk

  this._source.onend = function() {
    self.push(null);
  };
}

// _read will be called when the stream wants to pull more data
in
// the advisory size argument is ignored in this case.
SourceWrapper.prototype._read = function(size) {
  this._source.readStart();
};
```

Class: stream.Writable

`stream.Writable` 是一个被设计为需要实现底层的 `_write(chunk, encoding, callback)` 方法的抽象类。

请参阅上文的面向流消费者的API来了解如何在程序中消费流。以下解释了如果在你的程序中实现可写流。

new stream.Writable([options])

- **options Object**
  - highWaterMark Number `write()` 方法开始返回 `false` 的缓冲级别。默认为16kb，对于 `objectMode` 流则是 `16`
  - decodeStrings Boolean 是否在传递给 `write()` 方法前将字符串解码成 `Buffer` 。默认为 `true`
  - objectMode Boolean 是否 `write(anyObj)` 为一个合法操作。如果设置为 `true` 你可以写入任意数据而不仅是 `Buffer` 或字符串数据。默认为 `false`

在实现了 `Writable` 类的类中，请确保调用了 `Writable` 构造函数，这样缓冲设置才能被正确的初始化。

writable._write(chunk, encoding, callback)

- chunk Buffer | String 将要被写入的数据块。除非 `decodeStrings` 配置被设置为 `false` ，否则将一直是一个 `buffer`
- encoding String 如果数据块是一个字符串，那么这就是编码的类型。如果是一个 `buffer` ，那么则会忽略它
- callback Function 当你处理完给定的数据块后调用这个函数

所有的 `Writable` 流的实现都必须提供一个 `_write()` 方法来给底层资源传输数据。

这个函数不应该被直接调用。它应该被子类实现，并且仅被 `Writable` 类的内部方法调用。

回调函数使用标准的 `callback(error)` 模式来表示这个写操作成功或发生了错误。

如果构造函数选项中设置了 `decodeStrings` 标志，那么数据块将是一个字符串而不是一个 `Buffer` ，编码将会决定字符串的类型。这个是为了帮助处理编码字符串的实现。如果你没有明确地将 `decodeStrings` 选项设为 `false` ，那么你会安全地忽略 `encoding` 参数，并且数据块是 `Buffer` 形式。

这个函数有一个下划线前缀，因为它对于类是内部的，并应该直接被用户的程序调用。你应在你的拓展类里覆盖这个方法。

writable._writev(chunks, callback)

- chunks Array 将被写入的数据块数组。其中每一个数据都有如下格式： `{ chunk: ..., encoding: ... }`
- callback Function 当你处理完给定的数据块后调用这个函数

注意：这个函数不应该被直接调用。它应该被子类实现，并且仅被 `Writable` 类的内部方法调用。

这个函数对于你的实现是完全可选的。大多数情况下它是不必的。如果实现，它会被以所有滞留在写入队列中的数据块调用。

Class: stream.Duplex

一个“双工”流既是可读的，又是可写的。如TCP `socket` 连接。

注意，和你实现 `Readable` 或 `Writable` 流时一样， `stream.Duplex` 是一个被设计为需要实现底层的 `_read(size)` 和 `_write(chunk, encoding, callback)` 方法的抽象类。

由于 `JavaScript` 并不具备多继承能力，这个类是继承于 `Readable` 类，并寄生于 `Writable` 类。所以为了实现这个类，用户需要同时实现低级别的 `_read(n)` 方法和低级别的 `_write(chunk, encoding, callback)` 方法。

new stream.Duplex(options)

- **options Object** 同时传递给 `Writable` 和 `Readable` 构造函数。并且包含以下属性：
    - allowHalfOpen Boolean 默认为 `true` 。如果设置为 `false` ，那么流的可读的一端结束时可写的一端也会自动结束，反之亦然。
    - readableObjectMode Boolean 默认为 `false` ，为流的可读的一端设

  置 `objectMode` 。当 `objectMode` 为 `true` 时没有效果。
  - writableObjectMode Boolean 默认为 `false` ，为流的可写的一端设置 `objectMode` 。当 `objectMode` 为 `true` 时没有效果。

在实现了 `Duplex` 类的类中，请确保调用了 `Duplex` 构造函数，这样缓冲设置才能被正确的初始化。

Class: stream.Transform

“转换”流是一个输出于输入存在对应关系的双工流，如一个 `zilib` 流或一个 `crypto` 流。

输出和输出并不需要有相同的大小，相同的数据块数或同时到达。例如，一个哈希流只有一个单独数据块的输出当输入结束时。一个 `zlib` 流的输出比其输入小得多或大得多。

除了实现 `_read()` 方法和 `_write()` 方法，转换流还必须实现 `_transform()` 方法，并且可选地实现 `_flush()` 方法（参阅下文）。

new stream.Transform([options])

- options Object 同时传递给 `Writable` 和 `Readable` 构造函数。

在实现了 `Transform` 类的类中，请确保调用了 `Transform` 构造函数，这样缓冲设置才能被正确的初始化。

transform._transform(chunk, encoding, callback)

- chunk Buffer | String 将要被写入的数据块。除非 `decodeStrings` 配置被设置为 `false` ，否则将一直是一个 `buffer`
- encoding String 如果数据块是一个字符串，那么这就是编码的类型。如果是一个buffer，那么则会忽略它
- callback Function 当你处理完给定的数据块后调用这个函数

这个函数不应该被直接调用。它应该被子类实现，并且仅被 `Transform` 类的内部方法调用。

所有 `Transform` 流的实现都必须提供一个 `_transform` 方法来接受输入和产生输出。

在 `Transform` 类中， `_transform` 可以做需要做的任何事，如处理需要写入的字节，将它们传递给可写端，异步I/O，等等。

调用 `transform.push(outputChunk)` 0次或多次来从输入的数据块产生输出，取决于你想从这个数据块中输出多少数据作为结果。

仅当目前的数据块被完全消费后，才会调用回调函数。注意，对于某些特殊的输入可能会没有输出。如果你将数据作为第二个参数传入回调函数，那么数据将被传递给 `push` 方法。换句话说，下面的两个例子是相等的：

```
transform.prototype._transform = function (data, encoding, callb
ack) {
  this.push(data);
  callback();
}

transform.prototype._transform = function (data, encoding, callb
ack) {
  callback(null, data);
}
```

这个函数有一个下划线前缀，因为它对于类是内部的，并应该直接被用户的程序调用。你应在你的拓展类里覆盖这个方法。

transform._flush(callback)

- callback Function 当你排空了所有剩余数据后，这个回调函数会被调用

注意：这个函数不应该被直接调用。它应该被子类实现，并且仅被 `Transform` 类的内部方法调用。

在一些情景中，你的转换操作需要在流的末尾多发生一点点数据。例如，一个 `Zlib` 压缩流会存储一些内部状态以便它能优化压缩输出。但是在最后，它需要尽可能好得处理这些留下的东西来使数据完整。

在这种情况中，您可以实现一个 `_flush` 方法，它会在最后被调用，在所有写入数据被消费、但在触发 `end` 表示可读端到达末尾之前。和 `_transform` 一样，只需在写入操作完成时适当地调用 `transform.push(chunk)` 零或多次。

这个函数有一个下划线前缀，因为它对于类是内部的，并应该直接被用户的程序调用。你应在你的拓展类里覆盖这个方法。

Events: 'finish' 和 'end'

`finish` 和 `end` 事件分别来自于父类 `Writable` 和 `Readable` 。 `finish` 事件在 `end()` 方法被调用以及所有的输入被 `_transform` 方法处理后触发。 `end` 事件在所有的在 `_flush` 方法的回调函数被调用后的数据被输出后触发。

Example: SimpleProtocol 解释器 v2

上文中的简单协议解释器可以简单地通过高级别的 `Transform` 流更好地实现。与上文例子中的 `parseHeader` 和 `SimpleProtocol v1` 相似。

在这个例子中，没有从参数中提供输入，然后将它导流至解释器中，这更符合 `node.js` 的使用习惯。

```
var util = require('util');
var Transform = require('stream').Transform;
util.inherits(SimpleProtocol, Transform);

function SimpleProtocol(options) {
  if (!(this instanceof SimpleProtocol))
    return new SimpleProtocol(options);

  Transform.call(this, options);
  this._inBody = false;
  this._sawFirstCr = false;
  this._rawHeader = [];
  this.header = null;
}

SimpleProtocol.prototype._transform = function(chunk, encoding,
done) {
  if (!this._inBody) {
    // check if the chunk has a \n\n
    var split = -1;
    for (var i = 0; i < chunk.length; i++) {
      if (chunk[i] === 10) { // '\n'
```

```
        if (this._sawFirstCr) {
          split = i;
          break;
        } else {
          this._sawFirstCr = true;
        }
      } else {
        this._sawFirstCr = false;
      }
    }

    if (split === -1) {
      // still waiting for the \n\n
      // stash the chunk, and try again.
      this._rawHeader.push(chunk);
    } else {
      this._inBody = true;
      var h = chunk.slice(0, split);
      this._rawHeader.push(h);
      var header = Buffer.concat(this._rawHeader).toString();
      try {
        this.header = JSON.parse(header);
      } catch (er) {
        this.emit('error', new Error('invalid simple protocol da
ta'));
        return;
      }
      // and let them know that we are done parsing the header.
      this.emit('header', this.header);

      // now, because we got some extra data, emit this first.
      this.push(chunk.slice(split));
    }
  } else {
    // from there on, just provide the data to our consumer as-i
s.
    this.push(chunk);
  }
  done();
};
```

```
// Usage:
// var parser = new SimpleProtocol();
// source.pipe(parser)
// Now parser is a readable stream that will emit 'header'
// with the parsed header data.
```

Class: stream.PassThrough

这是一个 `Transform` 流的实现。将输入的流简单地传递给输出。它的主要目的是用来演示和测试，但它在某些需要构建特殊流的情况下可能有用。

简化的构造器API

可以简单的构造流而不使用继承。

这可以通过调用合适的方法作为构造函数和参数来实现：

例子：

Readable

```
var readable = new stream.Readable({
  read: function(n) {
    // sets this._read under the hood
  }
});
```

Writable

```
var writable = new stream.Writable({
  write: function(chunk, encoding, next) {
    // sets this._write under the hood
  }
});

// or

var writable = new stream.Writable({
  writev: function(chunks, next) {
    // sets this._writev under the hood
  }
});
```

Duplex

```
var duplex = new stream.Duplex({
  read: function(n) {
    // sets this._read under the hood
  },
  write: function(chunk, encoding, next) {
    // sets this._write under the hood
  }
});

// or

var duplex = new stream.Duplex({
  read: function(n) {
    // sets this._read under the hood
  },
  writev: function(chunks, next) {
    // sets this._writev under the hood
  }
});
```

Transform

```
var transform = new stream.Transform({
  transform: function(chunk, encoding, next) {
    // sets this._transform under the hood
  },
  flush: function(done) {
    // sets this._flush under the hood
  }
});
```

流：内部细节

缓冲

`Writable` 流和 `Readable` 流都会分别在一个内部的叫 `_writableState.buffer` 或 `_readableState.buffer` 的对象里缓冲数据。

潜在的被缓冲的数据量取决于被传递给构造函数的 `highWaterMark` 参数。

在 `Readable` 流中，当其的实现调用 `stream.push(chunk)` 时就会发生缓冲。如果流的消费者没有调用 `stream.read()` ，那么数据就会保留在内部队列中直到它被消费。

在 `Writable` 流中，当用户重复调用 `stream.write(chunk)` 时就会发生缓冲，甚至是当 `write()` 返回 `false` 时。

流，尤其是 `pipe()` 方法的初衷，是限制数据的滞留量在一个可接受的水平，这样才使得不同传输速度的来源和目标不会淹没可用的内存。

stream.read(0)

在一些情况下，你想不消费任何数据而去触发一次底层可读流机制的刷新。你可以调用 `stream.read(0)` ，它总是返回 `null` 。

如果内部的读缓冲量在 `highWaterMark` 之下，并且流没有正在读取，那么调用 `read(0)` 将会触发一次低级别的 `_read` 调用。

几乎永远没有必须这么做。但是，你可能会在 `node.js` 的 `Readable` 流类的内部代码的几处看到这个。

stream.push('')

推入一个0字节的字符串或 `Buffer` （不处于对象模式）有一个有趣的副作用。因为这是一个 `stream.push()` 的调用，它将会结束读取进程。但是，它不添加任何数据到可读缓冲中，所以没有任何用户可消费的数据。

在极少的情况下，你当下没有数据可以提供，但你的消费者同过调用 `stream.read(0)` 来得知合适再次检查。在这样的情况下，你可以调用 `stream.push('')` 。

至今为止，这个功能的唯一使用之处是在 `tls.CryptoStream` 类中，它将在 `node.js` 的1.0版本中被废弃。如果你发现你不得不使用 `stream.push('')` ，请考虑使用另外的方式。因为这几乎表示发生了某些可怕的错误。

与旧版本的 `Node.js` 的兼容性

在 `Node.js` 的0.10版本之前，可读流接口非常简单，并且功能和功用都不强。

- `data` 事件会立刻触发，而不是等待你调用 `read()` 方法。如果你需要进行一些 `I/O` 操作来决定是否处理数据，那么你只能将数据存储在某些缓冲区中以防数据流失。
- `pause()` 仅供查询，并不保证生效。这意味着你还是要准备接收 `data` 事件在流已经处于暂停模式中时。

在 `node.js` v1.0 和 `Node.js` v0.10中，下文所述的 `Readable` 类添加进来。为了向后兼容性，当一个 `data` 事件的监听器被添加时或 `resume()` 方法被调用时，可读流切换至流动模式。其作用是，即便您不使用新的 `read()` 方法和 `readable` 事件，您也不必担心丢失数据块。

大多数程序都会保持功能正常，但是，以下有一些边界情况：

- 没有添加任何 `data` 事件
- 从未调用 `resume()` 方法
- 流没有被导流至任何可写的目标

例如，考虑以下代码：

```
// WARNING!  BROKEN!
net.createServer(function(socket) {

  // we add an 'end' method, but never consume the data
  socket.on('end', function() {
    // It will never get here.
    socket.end('I got your message (but didnt read it)\n');
  });

}).listen(1337);
```

在 `Node.js` v0.10前，到来的信息数据会被简单地丢弃。但是在 `node.js` v1.0和 `Node.js` v0.10后， `socket` 会被永远暂停。

解决方案是调用 `resume()` 方法来开启数据流：

```
// Workaround
net.createServer(function(socket) {

  socket.on('end', function() {
    socket.end('I got your message (but didnt read it)\n');
  });

  // start the flow of data, discarding it.
  socket.resume();

}).listen(1337);
```

除了新的 `Readable` 流切换至流动模式之外，在v0.10之前的流可以被使用 `wrap()` 方法包裹。

对象模式

通常情况下，流仅操作字符串和 `Buffer` 。

处于对象模式中的流除了 `Buffer` 和字符串外，还能读出普通的 `JavaScript` 值。

处于对象模式中的可读流在调用 `stream.read(size)` 后只会返回单个项目，不论 `size` 参数是什么。

处于对象模式中的可写流总是忽略 `stream.write(data, encoding)` 中的 `encoding` 参数。

对于处于对象模式中的流，特殊值 `null` 仍然保留它的特殊意义。也就是说，对于对象模式的可读流， `stream.read()` 返回一个 `null` 仍意味着没有更多的数据了，并且 `stream.push(null)` 会发送一个文件末端信号（ `EOF` ）。

核心 `node.js` 中没有流是对象模式的。这个模式仅仅供用户的流库使用。

你应当在子类的构造函数的 `options` 参数对象中设置对象模式。在流的过程中设置对象模式时不安全的。

对于双工流，可以分别得通过 `readableObjectMode` 和 `writableObjectMode` 设置可读端和可写端。这些配置可以被用来通过转换流实现解释器和序列化器。

```
var util = require('util');
var StringDecoder = require('string_decoder').StringDecoder;
var Transform = require('stream').Transform;
util.inherits(JSONParseStream, Transform);

// Gets \n-delimited JSON string data, and emits the parsed obje
cts
function JSONParseStream() {
  if (!(this instanceof JSONParseStream))
    return new JSONParseStream();

  Transform.call(this, { readableObjectMode : true });

  this._buffer = '';
  this._decoder = new StringDecoder('utf8');
}

JSONParseStream.prototype._transform = function(chunk, encoding,
 cb) {
  this._buffer += this._decoder.write(chunk);
  // split on newlines
  var lines = this._buffer.split(/\r?\n/);
  // keep the last partial line buffered
```

```
  this._buffer = lines.pop();
  for (var l = 0; l < lines.length; l++) {
    var line = lines[l];
    try {
      var obj = JSON.parse(line);
    } catch (er) {
      this.emit('error', er);
      return;
    }
    // push the parsed object out to the readable consumer
    this.push(obj);
  }
  cb();
};

JSONParseStream.prototype._flush = function(cb) {
  // Just handle any leftover
  var rem = this._buffer.trim();
  if (rem) {
    try {
      var obj = JSON.parse(rem);
    } catch (er) {
      this.emit('error', er);
      return;
    }
    // push the parsed object out to the readable consumer
    this.push(obj);
  }
  cb();
};
```

StringDecoder

稳定度: 2 - 稳定

通过 `require('string_decoder')` 来使用这个模块。 `StringDecoder` 解码一个 `buffer` 为一个字符串。它是一个 `buffer.toString()` 的简单接口，但是提供了utf8的额外支持。

```
var StringDecoder = require('string_decoder').StringDecoder;
var decoder = new StringDecoder('utf8');

var cent = new Buffer([0xC2, 0xA2]);
console.log(decoder.write(cent));

var euro = new Buffer([0xE2, 0x82, 0xAC]);
console.log(decoder.write(euro));
```

Class: StringDecoder

接受一个单独的参数，即编码，默认为utf8。

decoder.write(buffer)

返回被解码的字符串。

decoder.end()

返回遗留在 `buffer` 中的所有末端字节。

Timers

稳定度: 3 - 锁定

所有的定时器函数都是全局的。当需要使用它们时，不必通过 `require()` 。

setTimeout(callback, delay[, arg][, ...])

在指定的延时（毫秒）后执行一次回调函数。返回一个可以被调用 `clearTimeout()` 的 `timeoutObject` 。可选的，你可以传递回调函数的参数。

需要注意的是，你的回调函数可以不会在精确的在指定的毫秒延时后执行 - `node.js` 对回调函数执行的精确时间以及顺序都不作保证。回调函数的执行点会尽量接近指定的延时。

clearTimeout(timeoutObject)

阻止一个 `timeout` 的触发。

setInterval(callback, delay[, arg][, ...])

在每次到达了指定的延时后，都重复执行回调函数。返回一个可以被调用 `clearInterval()` 的 `intervalObject` 。可选的，你可以传递回调函数的参数。

clearInterval(intervalObject)

阻止一个 `interval` 的触发。

unref()

`setTimeout` 和 `setInterval` 的返回值也有一个 `timer.unref()` 方法，这个方法允许你创建一个 当它是事件循环中的仅剩项时，它不会保持程序继续运行 的定时器。如果一个定时器已经被 `unref` ，再次调用 `unref` 不会有任何效果。

在 `setTimeout` 的情况下，当你调用 `unref` 时，你创建了一个将会唤醒事件循环的另一个定时器。创建太多这样的定时器会影响时间循环的性能 -- 请明智地使用。

ref()

如果你先前对一个定时器调用了 `unref()` ，你可以调用 `ref()` 来明确要求定时器要保持程序运行。如果一个定时器已经被 `ref` ，再次调用 `ref` 不会有任何效果。

setImmediate(callback[, arg][, ...])

在下一次I/O事件循环后，在 `setTimeout` 和 `setInterval` 前，"立刻"执行回调函数。返回一个可以被 `clearImmediate()` 的 `immediateObject` 。可选的，你可以传递回调函数的参数。

由 `setImmediate` 创建的回调函数会被有序地排队。每一次事件循环迭代时，整个回调函数队列都会被处理。如果你在一个执行中的回调函数里调用了 `setImmediate` ，那么这个 `setImmediate` 中的回调函数会在下一次事件循环迭代时被调用。

clearImmediate(immediateObject)

阻止一个 `immediate` 的触发。

TLS (SSL)

稳定度: 2 - 稳定

通过 `require('tls')` 来使用这个模块。

`tls` 模块使用OpenSSL来提供传输层的安全 和/或 安全 `socket` 层：已加密的流通信。

TLS/SSL是一种公/私钥架构。每个客户端和每个服务器都必须有一个私钥。一个私钥通过像如下的方式创建：

```
openssl genrsa -out ryans-key.pem 2048
```

所有的服务器和部分的客户端需要一个证书。证书是被CA签名或自签名的公钥。获取一个证书第一步是创建一个“证书签署请求（Certificate Signing Request）”（CSR）文件。通过：

```
openssl req -new -sha256 -key ryans-key.pem -out ryans-csr.pem
```

要通过CSR创建一个自签名证书，通过：

```
openssl x509 -req -in ryans-csr.pem -signkey ryans-key.pem -out
ryans-cert.pem
```

另外，你也可以把CSR交给一个CA请求签名。

为了完全向前保密（PFS），需要产生一个 迪菲-赫尔曼 参数：

```
openssl dhparam -outform PEM -out dhparam.pem 2048
```

创建 `.pfx` 或 `.p12` ，通过：

```
openssl pkcs12 -export -in agent5-cert.pem -inkey agent5-key.pem
 \
    -certfile ca-cert.pem -out agent5.pfx
```

- in: 证书
- inkey: 私钥
- certfile: 将所有 `CA certs` 串联在一个文件中，就像 `cat ca1-cert.pem ca2-cert.pem > ca-cert.pem` 。

客户端发起的重新协商攻击的减缓

TLS协议让客户端可以重新协商某些部分的TLS会话。不幸的是，会话重协商需要不相称的服务器端资源，这它可能成为潜在的DOS攻击。

为了减缓这种情况，重新协商被限制在了每10分钟最多3次。当超过阀值时， `tls.TLSSocket` 会触发一个错误。阀值是可以调整的：

- tls.CLIENT_RENEG_LIMIT: 重新协商限制，默认为 `3` 。

- tls.CLIENT_RENEG_WINDOW: 重新协商窗口（秒），默认为10分钟。

除非你知道你在做什么，否则不要改变默认值。

为了测试你的服务器，使用 `openssl s_client -connect address:port` 来连接它，然后键入 `R<CR>` （字母 `R` 加回车）多次。

NPN 和 SNI

`NPN` （下个协议协商）和SNI（服务器名称指示）都是TLS握手拓展，它们允许你：

- NPN - 通过多个协议（HTTP，SPDY）使用一个TLS服务器。
- SNI - 通过多个有不同的SSL证书的主机名来使用一个TLS服务器。

完全向前保密

术语“向前保密”或“完全向前保密”描述了一个密钥-协商（如密钥-交换）方法的特性。事实上，它意味着，甚至是当（你的）服务器的私钥被窃取了，窃取者也只能在他成功获得所有会话产生的密钥对时，才能解码信息。

它通过在每次握手中（而不是所有的会话都是同样的密钥）随机地产生用于密钥-协商的密钥对来实现。实现了这个技术的方法被称作“ephemeral”。

目前有两种普遍的方法来实现完全向前保密：

- DHE - 一个 迪菲-赫尔曼 密钥-协商 协议的 `ephemeral` 版本。
- ECDHE - 一个椭圆曲线 迪菲-赫尔曼 密钥-协商 协议的 `ephemeral` 版本。

`ephemeral` 方法可能有一些性能问题，因为密钥的生成是昂贵的。

tls.getCiphers()

返回支持的SSL加密器的名字数组。

例子：

```
var ciphers = tls.getCiphers();
console.log(ciphers); // ['AES128-SHA', 'AES256-SHA', ...]
```

tls.createServer(options[, secureConnectionListener])

创一个新的 `tls.Server` 实例。 `connectionListener` 参数被自动添加为 `secureConnection` 事件的监听器。 `options` 参数可以有以下属性：

- pfx: 一个包含 `PFX` 或 `PKCS12` 格式的私钥，加密凭证和CA证书的字符串或 `buffer` 。
- key: 一个带着 `PEM` 加密私钥的字符串（可以是密钥数组）（必选）。
- passphrase: 一个私钥或 `pfx` 密码字符串。
- cert: 一个包含了 `PEM` 格式的服务器证书密钥的字符串或 `buffer` （可以是 `cert` 数组）（必选）。
- ca: 一个 `PEM` 格式的受信任证书的字符串或 `buffer` 数组。如果它被忽略，将使用一些众所周知的“根”CA，像 `VeriSign` 。这些被用来授权连接。
- crl : 一个 `PEM` 编码的证书撤销列表（Certificate Revocation List）字符串或字符串列表。
- ciphers: 一个描述要使用或排除的加密器的字符串，通过 `:` 分割。默认的加密器套件是：

```
ECDHE-RSA-AES128-GCM-SHA256:
ECDHE-ECDSA-AES128-GCM-SHA256:
ECDHE-RSA-AES256-GCM-SHA384:
ECDHE-ECDSA-AES256-GCM-SHA384:
DHE-RSA-AES128-GCM-SHA256:
ECDHE-RSA-AES128-SHA256:
DHE-RSA-AES128-SHA256:
ECDHE-RSA-AES256-SHA384:
DHE-RSA-AES256-SHA384:
ECDHE-RSA-AES256-SHA256:
DHE-RSA-AES256-SHA256:
HIGH:
!aNULL:
!eNULL:
!EXPORT:
!DES:
!RC4:
!MD5:
!PSK:
!SRP:
!CAMELLIA
```

默认的加密器套件更倾向于 `Chrome's 'modern cryptography' setting` 的GCM加密器，也倾向于PFC的ECDHE和DHE加密器，它们提供了一些向后兼容性。

鉴于 `specific attacks affecting larger AES key sizes` ，所以更倾向于使用128位的AES而不是192和256位的AES。

旧的依赖于不安全的和弃用的RC4或基于DES的加密器（像IE6）的客户端将不能完成默认配置下的握手。如果你必须支持这些客户端， `TLS推荐规范` 可能提供了一个兼容的加密器套件。更多格式细节，参阅 `OpenSSL cipher list format documentation` 。

- ecdhCurve: 一个描述用于 `ECDH` 密钥协商的已命名的椭圆的字符串，如果要禁用 `ECDH` ，就设置为 `false` 。

默认值为 `prime256v1` （NIST P-256）。使用 `crypto.getCurves()` 来获取一个可用的椭圆列表。在最近的发行版中，运行 `openssl ecparam -list_curves` 命令也会展示所有可用的椭圆的名字和描述。

- dhparam: 一个包含了迪菲-赫尔曼参数的字符串或 `buffer` ，要求有完全向前保密。使用 `openssl dhparam` 来创建它。它的密钥长度需要大于等于1024字节，否则会抛出一个错误。强力推荐使用2048或更多位，来获取更高的安全性。如果参数被忽略或不合法，它会被默默丢弃并且 `DHE` 加密器将不可用。
- handshakeTimeout: 当SSL/TLS握手在这个指定的毫秒数后没有完成时，终止这个链接。默认为120秒。

当握手超时时， `tls.Server` 会触发一个 `clientError` 事件。

- honorCipherOrder : 选择一个加密器时，使用使用服务器的首选项而不是客户端的首选项。默认为 `true` 。
- requestCert: 如果设置为 `true` ，服务器将会向连接的客户端请求一个证书，并且试图验证这个证书。默认为 `true` 。
- rejectUnauthorized: 如果设置为 `true` ，服务器会拒绝所有没有在提供的CA列表中被授权的客户端。只有在 `requestCert` 为 `true` 时这个选项才有效。默认为 `false` 。
- NPNProtocols: 一个可用的 `NPN` 协议的字符串或数组（协议应该由它们的优先级被排序）。
- SNICallback(servername, cb): 当客户端支持 `SNI TLS` 扩展时，这个函数会被调用。这个函数会被传递两个参数：servername和cb。 `SNICallback` 必须执行 `cb(null, ctx)` , `ctx` 是一个 `SecureContext` 实例（你可以使用 `tls.createSecureContext(...)` 来获取合适的 `SecureContext` ）。如果 `SNICallback` 没有被提供 - 默认的有高层次API的回调函数会被使用（参阅下文）。
- sessionTimeout: 一个指定在TLS会话标识符和TLS会话门票（tickets）被服务器创建后的超时时间。更多详情参阅 `SSL_CTX_set_timeout` 。
- ticketKeys: 一个由16字节前缀，16字节hmac密钥，16字节AEC密钥组成的48字节 `buffer` 。你可以使用它在不同的 `tls` 服务器实例上接受 `tls` 会话门票。

注意：会在 `cluster` 模块工作进程间自动共享。

- sessionIdContext: 一个包含了会话恢复标识符的字符串。如果 `requestCert` 为 `true` ，默认值是通过命令行生成的MD5哈希值。否则，就将不提供默认值。
- secureProtocol: 将要使用的SSL方法，举例， `SSLv3_method` 将强制使用SSL v3。可用的值取决于OpenSSL的安装和 `SSL_METHODS` 常量中被定义的值。

下面是一个简单应答服务器的例子：

```
var tls = require('tls');
var fs = require('fs');

var options = {
  key: fs.readFileSync('server-key.pem'),
  cert: fs.readFileSync('server-cert.pem'),

  // This is necessary only if using the client certificate auth
entication.
  requestCert: true,

  // This is necessary only if the client uses the self-signed c
ertificate.
  ca: [ fs.readFileSync('client-cert.pem') ]
};

var server = tls.createServer(options, function(socket) {
  console.log('server connected',
              socket.authorized ? 'authorized' : 'unauthorized')
;
  socket.write("welcome!\n");
  socket.setEncoding('utf8');
  socket.pipe(socket);
});
server.listen(8000, function() {
  console.log('server bound');
});
```

或

```
var tls = require('tls');
var fs = require('fs');

var options = {
  pfx: fs.readFileSync('server.pfx'),

  // This is necessary only if using the client certificate auth
entication.
  requestCert: true,

};

var server = tls.createServer(options, function(socket) {
  console.log('server connected',
              socket.authorized ? 'authorized' : 'unauthorized')
;
  socket.write("welcome!\n");
  socket.setEncoding('utf8');
  socket.pipe(socket);
});
server.listen(8000, function() {
  console.log('server bound');
});
```

你可以通过 `openssl s_client` 来连接服务器：

```
openssl s_client -connect 127.0.0.1:8000
```

tls.connect(options[, callback])

tls.connect(port[, host][, options][, callback])

根据给定的 端口和主机（旧API）或 `options.port` 和 `options.host` 创建一个新的客户端连接。如果忽略了主机，默认为 `localhost` 。 `options` 可是一个含有以下属性的对象：

- host: 客户端应该连接到的主机。
- port: 客户端应该连接到的端口。
- socket: 根据给定的 `socket` 的来建立安全连接，而不是创建一个新的 `socket` 。如果这个选项被指定， `host` 和 `port` 会被忽略。
- path: 创建到 `path` 的unix `socket` 连接。如果这个选项被指定， `host` 和 `port` 会被忽略。
- pfx: 一个 `PFX` 或 `PKCS12` 格式的包含了私钥，证书和CA证书的字符串或 `buffer` 。
- key: 一个 `PEM` 格式的包含了客户端私钥的字符串或 `buffer` （可以是密钥的数组）。
- passphrase: 私钥或 `pfx` 的密码字符串。
- cert: 一个 `PEM` 格式的包含了证书密钥的字符串或 `buffer` （可以是密钥的数组）。
- ca: 一个 `PEM` 格式的受信任证书的字符串或 `buffer` 数组。如果它被忽略，将使用一些众所周知的CA，像 `VeriSign` 。这些被用来授权连接。
- ciphers: 一个描述了要使用或排除的加密器，由 `:` 分割。使用的默认加密器套件与 `tls.createServer` 使用的一样。
- rejectUnauthorized: 若被设置为 `true` ，会根据提供的CA列表来验证服务器证书。当验证失败时，会触发 `error` 事件； `err.code` 包含了一个OpenSSL错误码。默认为 `true` 。
- NPNProtocols: 包含支持的NPN协议的字符串或 `buffer` 数组。 `buffer` 必须有以下格式： `0x05hello0x05world` ，第一个字节是下一个协议名的长度（传递数组会更简单： `['hello', 'world']` ）。
- servername: `SNI` TLS 扩展的服务器名。
- checkServerIdentity(servername, cert): 为根据证书的服务器主机名检查提供了覆盖。必须在验证失败时返回一个错误，验证通过时返回 `undefined` 。
- secureProtocol: 将要使用的SSL方法，举例， `SSLv3_method` 将强制使用SSL v3。可用的值取决于OpenSSL的安装和 `SSL_METHODS` 常量中被定义的值。

- session: 一个 `Buffer` 实例，包含了TLS会话。

`callback` 参数会被自动添加为 `secureConnect` 事件的监听器。

`tls.connect()` 返回一个 `tls.TLSSocket` 对象。

以下是一个上述应答服务器的客户端的例子：

```
var tls = require('tls');
var fs = require('fs');

var options = {
  // These are necessary only if using the client certificate au
thentication
  key: fs.readFileSync('client-key.pem'),
  cert: fs.readFileSync('client-cert.pem'),

  // This is necessary only if the server uses the self-signed c
ertificate
  ca: [ fs.readFileSync('server-cert.pem') ]
};

var socket = tls.connect(8000, options, function() {
  console.log('client connected',
              socket.authorized ? 'authorized' : 'unauthorized')
;
  process.stdin.pipe(socket);
  process.stdin.resume();
});
socket.setEncoding('utf8');
socket.on('data', function(data) {
  console.log(data);
});
socket.on('end', function() {
  server.close();
});
```

或

```
var tls = require('tls');
var fs = require('fs');

var options = {
  pfx: fs.readFileSync('client.pfx')
};

var socket = tls.connect(8000, options, function() {
  console.log('client connected',
              socket.authorized ? 'authorized' : 'unauthorized')
;
  process.stdin.pipe(socket);
  process.stdin.resume();
});
socket.setEncoding('utf8');
socket.on('data', function(data) {
  console.log(data);
});
socket.on('end', function() {
  server.close();
});
```

Class: tls.TLSSocket

`net.Socket` 实例的包装，替换了内部 `socket` 的 读/写例程，来提供透明的对传入/传出数据 的 加密/解密。

new tls.TLSSocket(socket, options)

根据已存在的TCP `socket` ，构造一个新的 `TLSSocket` 对象。

`socket` 是一个 `net.Socket` 实例。

`options` 是一个可能包含以下属性的对象：

- secureContext: 一个可选的通过 `tls.createSecureContext( ... )` 得到的TLS内容对象。
- isServer: 如果为 `true` ，TLS `socket` 将会在服务器模式（server-mode）下被初始化。

- server: 一个可选的 `net.Server` 实例。
- requestCert: 可选，参阅 `tls.createSecurePair` 。
- rejectUnauthorized: 可选，参阅 `tls.createSecurePair` 。
- NPNProtocols: 可选，参阅 `tls.createServer` 。
- SNICallback: 可选，参阅 `tls.createServer` 。
- session: 可选，一个 `Buffer` 实例，包含了TLS会话。
- requestOCSP: 可选，如果为 `true` ， `OCSP` 状态请求扩展将会被添加到客户端 hello，并且 `OCSPResponse` 事件将会在建立安全通信前，于 `socket` 上触发。

tls.createSecureContext(details)

创建一个证书对象， `details` 有可选的以下值：

- pfx : 一个含有 `PFX` 或 `PKCS12` 编码的私钥，证书和CA证书的字符串或 `buffer` 。
- key : 一个含有 `PEM` 编码的私钥的字符串。
- passphrase : 一个私钥或 `pfx` 密码字符串。
- cert : 一个含有 `PEM` 加密证书的字符串。
- ca : 一个用来信任的 `PEM` 加密CA证书的字符串或字符串列表。
- crl : 一个 `PEM` 加密 `CRL` 的字符串或字符串列表。
- ciphers: 一个描述需要使用或排除的加密器的字符串。更多加密器的格式细节参阅 `http://www.openssl.org/docs/apps/ciphers.html#CIPHER_LIST_FORMAT` 。
- honorCipherOrder : 选择一个加密器时，使用使用服务器的首选项而不是客户端的首选项。默认为 `true` 。更多细节参阅 `tls` 模块文档。

如果没有指定 `ca` ，那么 `node.js` 将会使用 `http://mxr.mozilla.org/mozilla/source/security/nss/lib/ckfw/builtins/certdata.txt` 提供的默认公共可信任CA列表。

tls.createSecurePair([context][, isServer][, requestCert][, rejectUnauthorized])

根据两个流，创建一个新的安全对（secure pair）对象，一个是用来读/写加密数据，另一个是用来读/写明文数据。通常加密的数据是从加密数据流被导流而来，明文数据被用来作为初始加密流的一个替代。

- credentials: 一个通过 `tls.createSecureContext( ... )` 得到的安全内容对象。
- isServer: 一个表明了 是否这个 `tls` 连接应被作为一个服务器或一个客户端打开 的布尔值。
- requestCert: 一个表明了 是否服务器应该向连接的客户端请求证书 的布尔值。只应用于服务器连接。
- rejectUnauthorized: 一个表明了 是否服务器应该拒绝包含不可用证书的客户端的布尔值。只应用于启用了 `requestCert` 的服务器。

`tls.createSecurePair()` 返回一个带有 `cleartext` 和 `encrypted` 流 属性的对象。

注意： `cleartext` 和 `tls.TLSSocket` 有相同的API。

Class: SecurePair

由 `tls.createSecurePair` 返回。

Event: 'secure'

当 `SecurePair` 成功建立一个安全连接时， `SecurePair` 会触发这个事件

与检查服务器的 `secureConnection` 事件相似， `pair.cleartext.authorized` 必须被检查，来确认证书是否使用了合适的授权。

Class: tls.Server

这是一个 `net.Server` 的子类，并且与其有相同的方法。除了只接受源TCP连接，这个类还接受通过TLS或SSL加密的数据。

Event: 'secureConnection'

- function (tlsSocket) {}

当一个新连接被成功握手后，这个事件会被触发。参数是一个 `tls.TLSSocket` 实例。它拥有所有普通流拥有的事件和方法。

`socket.authorized` 是一个表明了 客户端是否通过提供的服务器CA来进行了认证 的布尔值。如果 `socket.authorized` 为 `false` ，那么 `socket.authorizationError` 将被设置用来描述授权失败的原因。一个不明显的但是值得提出的点：依靠TLS服务器的设定，未授权的连接可能会被接受。 `socket.npnProtocol` 是一个包含了被选择的NPN协议的字符串。 `socket.servernam` 是一个包含了通过SNI请求的服务器名的字符串。

Event: 'clientError'

- function (exception, tlsSocket) { }

当安全连接被建立之前，服务器触发了一个 `error` 事件 - 它会被转发到这里。

`tlsSocket` 是错误来自的 `tls.TLSSocket` 。

Event: 'newSession'

- function (sessionId, sessionData, callback) { }

在TLS会话创建时触发。可能会被用来在外部存储会话。 `callback` 必须最终被执行，否则安全连接将不会收到数据。

注意：这个事件监听器只会影响到它被添加之后建立的连接。

Event: 'resumeSession'

- function (sessionId, callback) { }

当客户端想要恢复先前的TLS会话时触发。事件监听器可能会在外部通过 `sessionId` 来寻找会话，并且在结束后调用 `callback(null, sessionData)` 。如果会话不能被恢复（例如没有找到），可能会调用 `callback(null, null)` 。调用 `callback(err)` 会关闭将要到来的连接并且销毁 `socket` 。

注意：这个事件监听器只会影响到它被添加之后建立的连接。

Event: 'OCSPRequest'

- function (certificate, issuer, callback) { }

当客户端发送一个证书状态请求时触发。你可以解释服务器当前的证书来获取OCSP url和证书id，并且在获取了OCSP响应后执行 `callback(null, resp)` ， `resp` 是一个 `Buffer` 实例。 `certificate` 和 `issuer` 都是一个 `Buffer` ，即主键和发起人证书的DER代表（DER-representations）。它们可以被用来获取OCSP证书id 和 OCSP末端url。

另外， `callback(null, null)` 可以被调用，意味着没有OCSP响应。

调用 `callback(err)` ，将会导致调用 `socket.destroy(err)` 。

典型的流程：

1. 客户端连接到服务器，然后发送一个 `OCSPRequest` 给它（通过 `ClientHello` 中扩展的状态信息）。
2. 服务器接受请求，然后执行 `OCSPRequest` 事件监听器（如果存在）。
3. 服务器通过证书或发起人抓取OCSP url，然后向CA发起一个OCSP请求。
4. 服务器从CA收到一个 `OCSPResponse` ，然后通过回调函数的参数将其返回给客户端。
5. 客户端验证响应，然后销毁 `socket` 或者进行握手。

注意： `issuer` 可以是 `null` ，如果证书是自签名的或 `issuer` 不在根证书列表之内（你可以通过 `ca` 参数提供一个 `issuer` ）。

注意：这个事件监听器只会影响到它被添加之后建立的连接。

注意：你可能想要使用一些如 `asn1.js` 的 `npm` 模块来解释证书。

server.listen(port[, hostname][, callback])

从指定的端口和主机名接收连接。如果 `hostname` 被忽略，服务器会在当IPv6可用时，接受任意IPv6地址（ `::` ）上的连接，否则为任意IPv4（ `0.0.0.0` ）上的。将 `port` 设置为 `0` 则会赋予其一个随机端口。

这个函数是异步的。最后一个参数 `callback` 会在服务器被绑定后执行。

更多信息请参阅 `net.Server` 。

server.close([callback])

阻止服务器继续接收新连接。这个函数是异步的，当服务器触发一个 `close` 事件时，服务器将最终被关闭。可选的，你可以传递一个回调函数来监听 `close` 事件。

server.address()

返回绑定的地址，服务器地址的协议族名和端口通过操作系统报告。更多信息请参阅 `net.Server.address()` 。

server.addContext(hostname, context)

添加安全内容，它将会在如果客户端请求的SNI主机名被传递的主机名匹配（可以使用通配符）时使用。 `context` 可以包含密钥，证书，CA 和/或 其他任何 `tls.createSecureContext` 的 `options` 参数的属性。

server.maxConnections

当服务器连接数变多时，设置这个值来拒绝连接。

server.connections

服务器上的当前连接数。

Class: CryptoStream

> 稳定度: 0 - 弃用。 使用 `tls.TLSSocket` 替代。

这是一个加密流。

cryptoStream.bytesWritten

一个底层 `socket` 的 `bytesWritten` 存取器的代理，它会返回写入 `socket` 的总字节数，包括TLS开销。

Class: tls.TLSSocket

这是一个 `net.Socket` 的包装，但是对写入的数据做了透明的加密，并且要求TLS协商。

这个实例实现了一个双工流接口。它有所有普通流所拥有的事件和方法。

Event: 'secureConnect'

在一个新连接成功握手后，这个事件被触发。无论服务器的证书被授权与否，这个监听器都会被调用。测试 `tlsSocket.authorized` 来验证服务器证书是否被一个指定CA所签名 取决于用户。如果 `tlsSocket.authorized === false` 那么错误可以从 `tlsSocket.authorizationError` 里被发现。如果 `NPN` 被使用，你可以通过 `tlsSocket.npnProtocol` 来检查已协商协议。

Event: 'OCSPResponse'

- function (response) { }

如果 `requestOCSP` 选项被设置，这个事件会触发。 `response` 是一个 `buffer` 对象，包含了服务器的OCSP响应。

习惯上， `response` 是一个来自服务器的CA（包含服务器的证书撤销状态）的已签名对象。

tlsSocket.encrypted

静态布尔变量，总是 `true` 。可能会被用来区分TLS `socket` 和普通的 `socket` 。

tlsSocket.authorized

如果对等（peer）证书通过一个指定的CA被签名，那么这个值为 `true` 。否则为 `false` 。

tlsSocket.authorizationError

对等（peer）的证书没有被验证的原因。这个值只在 `tlsSocket.authorized === false` 时可用。

tlsSocket.getPeerCertificate([detailed])

返回了一个代表了对等证书的对象。返回的对象有一些属性与证书的属性一致。如果 `detailed` 参数被设置为 `true` ， `issuer` 属性的完整链都会被返回，如果为 `false` ，只返回不包含 `issuer` 属性的顶端的证书。

例子：

```
{ subject:
   { C: 'UK',
     ST: 'Acknack Ltd',
     L: 'Rhys Jones',
     O: 'node.js',
     OU: 'Test TLS Certificate',
     CN: 'localhost' },
  issuerInfo:
   { C: 'UK',
     ST: 'Acknack Ltd',
     L: 'Rhys Jones',
     O: 'node.js',
     OU: 'Test TLS Certificate',
     CN: 'localhost' },
  issuer:
   { ... another certificate ... },
  raw: < RAW DER buffer >,
  valid_from: 'Nov 11 09:52:22 2009 GMT',
  valid_to: 'Nov  6 09:52:22 2029 GMT',
  fingerprint: '2A:7A:C2:DD:E5:F9:CC:53:72:35:99:7A:02:5A:71:38:
52:EC:8A:DF',
  serialNumber: 'B9B0D332A1AA5635' }
```

如果 `peer` 没有提供一个证书，那么会返回 `null` 或空对象。

tlsSocket.getCipher()

返回一个代表了当前连接的加密器名和SSL/TLS协议版本的对象。

例子： `{ name: 'AES256-SHA', version: 'TLSv1/SSLv3' }`

参阅 `http://www.openssl.org/docs/ssl/ssl.html#DEALING_WITH_CIPHERS` 中 `SSL_CIPHER_get_name()` 和 `SSL_CIPHER_get_version()` 。

tlsSocket.renegotiate(options, callback)

初始化TLS重新协商过程。 `optios` 可以包含以下属性： `rejectUnauthorized` ， `requestCert` （详情参阅 `tls.createServer` ）。一旦重协商成功， `callback(err)` 会带着 `err` 为 `null` 执行。

注意：可以被用来请求对等（peer）证书在安全连接建立之后。

另一个注意点：当作为服务器运行时， `socekt` 在 `handshakeTimeout` 超时后，会带着一个错误被销毁。

tlsSocket.setMaxSendFragment(size)

设置TLS碎片大小的最大值（默认最大值为 `16384` ，最小值为 `512` ）。若设置成功返回 `true` ，否则返回 `false` 。

更小的碎片大小来减少客户端的缓冲延迟：大的碎片通过TLS层缓冲，直到收到全部的碎片并且它的完整性被验证；大碎片可能会跨越多次通信，并且可能会被报文丢失和重新排序所延迟。但是，更小的碎片增加了额外的TLS框架字节和CPU开销，可能会减少总体的服务器负载。

tlsSocket.getSession()

返回 `ASN.1` 编码的TLS会话，如果没有被协商，返回 `undefined` 。可以被用在重新连接服务器时，加速握手的建立。

tlsSocket.getTLSTicket()

注意：仅在客户端TLS `socket` 中工作。仅在调试时有用，因为会话重新使用了给 `tls.connect` 提供的 `session` 选项。

返回TLS会话门票（ticket），如果没有被协商，返回 `undefined` 。

tlsSocket.address()

返回绑定的地址，协议族名和端口由底层系统报告。返回一个含有三个属性的对象，例如： `{ port: 12346, family: 'IPv4', address: '127.0.0.1' }` 。

tlsSocket.remoteAddress

代表了远程IP地址的字符串。例子：`'74.125.127.100'` 或 `'2001:4860:a005::68'` 。

tlsSocket.remoteFamily

代表了远程IP协议族的字符串。 `'IPv4'` 或 `'IPv6'` 。

tlsSocket.remotePort

代表了远程端口数字。例子： `443` 。

tlsSocket.localAddress

代表了本地IP地址的字符串。

tlsSocket.localPort

代表了本地端口的数字。

TTY

Stability: 2 - Stable

`tty` 模块主要提供了 `tty.ReadStream` 和 `tty.WriteStream` 这两个类。大多数情况下，你都不需要直接使用这个模块。

当 `node.js` 检测到它运行于TTY上下文中，那么 `process.stdin` 将会是一个 `tty.ReadStream` 实例， `process.stdout` 将会是一个 `tty.WriteStream` 实例。测试 `node.js` 是否运行在TTY上下文中的一个比较好的办法是检查 `process.stdout.isTTY` ：

```
$ iojs -p -e "Boolean(process.stdout.isTTY)"
true
$ iojs -p -e "Boolean(process.stdout.isTTY)" | cat
false
```

tty.isatty(fd)

如果 `fd` 关联了终端，就返回 `true` ，反之返回 `false` 。

tty.setRawMode(mode)

已弃用。使用 `tty.ReadStream#setRawMode()` （如 `process.stdin.setRawMode()` ）代替。

Class: ReadStream

一个 `net.Socket` 子类，代表了一个TTY中的可读部分。一般情况下，在任何 `node.js` 程序（仅当 `isatty(0)` 为 `true` 时）中， `process.stdin` 将是仅有的 `tty.ReadStream` 实例。

rs.isRaw

一个被初始化为 `false` 的布尔值。它代表了 `tty.ReadStream` 实例的“原始”状态。

rs.setRawMode(mode)

`mode` 必须为 `true` 或 `false` 。它设定 `tty.ReadStream` 的属性表现得像原始设备或默认值。 `isRaw` 将会被设置为结果模式（resulting mode）。

Class: WriteStream

一个 `net.Socket` 子类，代表了一个TTY中的可写部分。一般情况下，在任何 `node.js` 程序（仅当 `isatty(1)` 为 `true` 时）中， `process.stdout` 将是仅有的 `tty.WriteStream` 实例。

ws.columns

一个表示了TTY当前拥有列数的数字。这个属性会通过 `resize` 事件被更新。

ws.rows

一个表示了TTY当前拥有行数的数字。这个属性会通过 `resize` 事件被更新。

Event: 'resize'

- function () {}

当列属性或行属性被改变时，通过 `refreshSize()` 被触发。

```
process.stdout.on('resize', function() {
  console.log('screen size has changed!');
  console.log(process.stdout.columns + 'x' + process.stdout.rows
);
});
```

UDP / Datagram Sockets

稳定度: 2 - 稳定

数据报 `socket` 通过 `require('dgram')` 使用。

重要提示： `dgram.Socket#bind()` 的表现在v0.10中被改变，并且现在总是异步的，如果你有像这样的代码：

```
var s = dgram.createSocket('udp4');
s.bind(1234);
s.addMembership('224.0.0.114');
```

你必须改成这样：

```
var s = dgram.createSocket('udp4');
s.bind(1234, function() {
  s.addMembership('224.0.0.114');
});
```

dgram.createSocket(type[, callback])

- type String. `'udp4'` 或 `'udp6'` ，两者之一
- callback Function. 可选，会被添加为 `message` 事件的监听器
- Returns: `socket` 对象

创建一个指定类型的数据报 `socket` 。可用类型是udp4和udp6.

接受一个可选的回调函数，它会被自动添加为 `message` 事件的监听器。

如果你想要接收数据报，调用 `socket.bind()` 。 `socket.bind()` 将会到 `所有网络接口` 地址中的一个随机端口（不论udp4和upd6 `socket` ，它都可以正常工作）。你可以从 `socket.address().address` 和 `socket.address().port` 中获取地址和端口。

dgram.createSocket(options[, callback])

- options Object
- callback Function. 会被添加为 `message` 事件的监听器
- Returns: `socket` 对象

`options` 对象必须包含一个 `type` 属性，可是udp4或udp6。还有一个可选的 `reuseAddr` 布尔值属性。

当 `reuseAddr` 为 `true` 时， `socket.bind()` 会重用地址，甚至是当另一个进程已经在这之上绑定了一个 `socket` 时。默认为 `false` 。

接受一个可选的回调函数，它会被自动添加为 `message` 事件的监听器。

如果你想要接收数据报，调用 `socket.bind()` 。 `socket.bind()` 将会到 `所有网络接口` 地址中的一个随机端口（不论udp4和upd6 `socket` ，它都可以正常工作）。你可以从 `socket.address().address` 和 `socket.address().port` 中获取地址和端口。

Class: dgram.Socket

`dgram.Socket` 类封装了数据报的功能。它必须被 `dgram.createSocket(...)` 创建。

Event: 'message'

- msg Buffer object. 消息
- rinfo Object. 远程地址信息

当在 `socket` 中一个新的数据报可用时触发。 `msg` 是一个 `buffer` 并且 `rinfo` 是一个包含发送者地址信息的对象：

```
socket.on('message', function(msg, rinfo) {
  console.log('Received %d bytes from %s:%d\n',
              msg.length, rinfo.address, rinfo.port);
});
```

Event: 'listening'

当一个 `socket` 开始监听数据报时触发。在UDP `socket` 被创建时触发。

Event: 'close'

在一个 `socket` 通过 `close()` 被关闭时触发。这个 `socket` 中不会再触发新的 `message` 事件。

Event: 'error'

- exception Error object

当错误发生时触发。

socket.send(buf, offset, length, port, address[, callback])

- buf Buffer object or string. 要被发送的信息。
- offset Integer. 信息在 `buffer` 里的初始偏移位置。
- length Integer. 信息的字节数。
- port Integer. 目标端口。
- address String. 目标主机或IP地址。
- callback Function. 可选，当信息被发送后调用。

对于UDP `socket` ，目标端口和地址都必须被指定。 `address` 参数需要提供一个字符串，并且它会被DNS解析。

如果 `address` 被忽略，或者是一个空字符串。将会使用 `'0.0.0.0'` 或 `'::0'` 。这取决于网络配置，这些默认值 可能会 或 可能不会 正常工作；所以最好还是明确指定目标地址。

如果一个 `socket` 先前没有被调用 `bind` 来绑定，它将会赋于一个随机端口数并且被绑定到“所有网络接口”地址（udp4 `socket` 为 `'0.0.0.0'` ，udp6则为 `'::0'` ）。

一个可选的回调函数可以被指定，用来检测DNS错误，或决定重用 `buf` 对象是否安全。注意，DNS查找至少会延迟一个事件循环。唯一能确定数据报被发送的方法就是使用一个回调函数。

出于对多字节字符的考虑， `offset` 和 `length` 将会根据字节长度而不是字符位置被计算。

一个向 `localhost` 上的一个随机端口发送UDP报文的例子：

```
var dgram = require('dgram');
var message = new Buffer("Some bytes");
var client = dgram.createSocket("udp4");
client.send(message, 0, message.length, 41234, "localhost", func
tion(err) {
  client.close();
});
```

UDP数据报大小的注意事项

IPv4/v6数据报的最大大小取决于 `MTU` （最大传输单位），和 `Payload Length` 字段大小。

- `Payload Length` 是16字节宽的，意味着一个正常的负载不能超过64K 八位字节，包括网络头和数据（65,507 字节 = 65,535 − 8 字节 UDP 头 − 20 字节 IP 头）；对于环回接口总是 `true` ，但是如此大的数据报对于大多数主机和网络来说都是不现实的。
- `MTU` 是指定的链路层技术支持的报文的最大大小。对于所有连接，IPv4允许最小 `MTU` 为68八位字节，而推荐的IPv4 `MTU` 是576（通常作为拨号类应用的推荐 `MTU` ），无论它们是完整的还是以碎片形式到达。
- 对于IPv6，最小MTU是1280八位字节，但是，允许的最小 `buffer` 重组大小是1500八位字节。68八位字节非常小，所以大多数的当前链路层技术的最小 `MTU` 都是1500（如 `Ethernet` ）。

注意，不可能提前知道一个报文可能经过的每一个连接 `MTU` ，并且通常不能发送一个大于（接收者） `MTU` 的数据报（报文会被默默丢弃，不会通知源头：这个数据没有到达已定的接收方）。

socket.bind(port[, address][, callback])

- port Integer
- address String, 可选
- callback Function 可选，没有参数。当绑定完毕后触发。

对于UDP `socket` ，监听一个具名的端口和一个可选的地址上的数据报。如果 `address` 没有被指定，操作系统将会试图监听所有端口。在绑定完毕后， `listening` 事件会被吃法，并且回调函数（如果指定了）会被调用。同时指

定 `listening` 事件的监听器和 `callback` 没有危险，但是不是很有用。

一个绑定的数据报 `socket` 将会保持 `node.js` 进程的运行，来接受数据报。

如果绑定失败，一个 `error` 事件会产生。极少数情况下（例如绑定一个关闭的 `socket` ），这个方法会抛出一个错误。

一个监听41234端口的UDP服务器：

```
var dgram = require("dgram");

var server = dgram.createSocket("udp4");

server.on("error", function (err) {
  console.log("server error:\n" + err.stack);
  server.close();
});

server.on("message", function (msg, rinfo) {
  console.log("server got: " + msg + " from " +
    rinfo.address + ":" + rinfo.port);
});

server.on("listening", function () {
  var address = server.address();
  console.log("server listening " +
      address.address + ":" + address.port);
});

server.bind(41234);
// server listening 0.0.0.0:41234
```

socket.bind(options[, callback])

- **options Object** - 必选，支持以下属性：
    - port Number - 必须
    - address String - 可选
    - exclusive Boolean - 可选
- callback Function - 可选

`options` 的 `prot` 和 `address` 属性，以及可选的回调函数，与 `socket.bind(port, [address], [callback])` 中它们的表现一致。

如 `exclusive` 为 `false` （默认），那么集群的工作进程将会使用相同的底层句柄，允许共享处理连接的职责。当为 `true` 时，句柄不被共享，企图共享端口会导致一个错误。一个监听一个 `exclusive` 端口的例子：

```
socket.bind({
  address: 'localhost',
  port: 8000,
  exclusive: true
});
```

socket.close([callback])

关闭底层 `socket` ，并且停止监听新数据。如果提供了回调函数，它会被添加为 `close` 事件的监听器。

socket.address()

返回一个包含 `socket` 地址信息的对象。对于UDP `socket` ，这个对象将会包含 `address` ， `family` 和 `port` 。

socket.setBroadcast(flag)

- flag Boolean

设置或清除 `SO_BROADCAST` `socket` 设置。当这个选项被设置，UDP报文将会被送至本地接口的广播地址。

socket.setTTL(ttl)

- ttl Integer

设置 `IP_TTL` `socket` 选项。 `TTL` 的意思是“生存时间（Time to Live）”，但是在这里的上下文中，它值一个报文通过的IP跃点数。每转发报文的路由或网关都会递减 `TTL` 。如果 `TTL` 被一个路由递减为 `0` ，它将不再被转发。改变 `TTL` 值常用于网络探测器或多播。

`setTTL()` 的参数是一个 `1` 到 `225` 之间的跃点数。多数系统中的默认值为 `64` 。

socket.setMulticastTTL(ttl)

- ttl Integer

设置 `IP_MULTICAST_TTL` `socket` 选项。 `TTL` 的意思是“生存时间（Time to Live）”，但是在这里的上下文中，它值一个报文通过的IP跃点数，特别是组播流量。每转发报文的路由或网关都会递减 `TTL` 。如果 `TTL` 被一个路由递减为 `0` ，它将不再被转发。

`setMulticastTTL()` 的参数是一个 `0` 到 `225` 之间的跃点数。多数系统中的默认值为 `1` 。

socket.setMulticastLoopback(flag)

- flag Boolean

设置或清除 `IP_MULTICAST_LOOP` `socket` 选项。当这个选项被设置，组播报文也将会在本地接口上接收。

socket.addMembership(multicastAddress[, multicastInterface])

- multicastAddress String
- multicastInterface String, 可选

告诉内核加入一个组播分组，通过 `IP_ADD_MEMBERSHIP` `socket` 选项。

如果 `multicastInterface` 没有被指定，那么操作系统将会尝试加入成为所有可用的接口的成员。

socket.dropMembership(multicastAddress[, multicastInterface])

- multicastAddress String
- multicastInterface String, 可选

与 `addMembership` 相反 - 告诉内核离开一个组播分组，通过 `IP_DROP_MEMBERSHIP` `socket` 选项。当 `socket` 被关闭或进程结束时，它会被内核自动调用。所以大多数应用不需要亲自调用它。

如果 `multicastInterface` 没有被指定，那么操作系统将会尝试脱离所有可用的接口。

socket.unref()

在一个 `socket` 上调用 `unref` 将会在它是事件系统中唯一活跃的 `socket` 时，允许程序退出。如果 `socket` 已经被 `unref` ，再次调用将不会有任何效果。

返回一个 `socket` 。

socket.ref()

与 `unref` 相反，在一个先前被 `unref` 的 `socket` 上调用 `ref` ，那么在它是唯一的剩余的 `socket` （默认行为）时，将不允许程序退出。如果 `socket` 已经被 `ref` ，再次调用将不会有任何效果。

返回一个 `socket` 。

URL

稳定度: 2 - 稳定

这个模块提供了URL解析和解释的工具。通过 `require('url')` 使用它。

解释URL为一个含有以下部分或全部属性的对象，依赖于它们是否在URL字符串中存在。任何不存在的部分都不会出现在解释后的对象中。一个下面URL的例子：

`'http://user:pass@host.com:8080/p/a/t/h?query=string#hash'`

- href: 最初传递的全部URL。协议和主机都是小写的。

例子： `'http://user:pass@host.com:8080/p/a/t/h?query=string#hash'`

- protocol: 请求的协议，小写。

例子： `'http:'`

- slashes: 协议要求冒号后有斜杠。

例子： `true` 或 `false`

- host: URL的所有主机部分，包括端口，小写。

例子： `'host.com:8080'`

- auth: URL的认证信息部分。

例子： `'user:pass'`

- hostname: 小写的主机名部分。

例子： `'host.com'`

- port: 主机部分的端口号。

例子： `'8080'`

- pathname: URL的路径部分，在主机之后，在查询之前，包括最前面的斜杠，如果存在的话。不提供解码。

例子： `'/p/a/t/h'`

- search: URL的“查询字符串”部分，包括前导的问号标志。

例子： `'?query=string'`

- path: 路径和查询的连接体。不提供解码。

例子： `'/p/a/t/h?query=string'`

- query: 查询字符串的“参数”部分，或查询字符串被解释后的对象。

例子： `'query=string'` 或 `{'query':'string'}`

- hash: URL的“碎片”部分，包括英镑符号。

例子： `'#hash'`

以下是URL模块提供的方法：

url.parse(urlStr[, parseQueryString][, slashesDenoteHost])

接收一个URL字符串，然后返回一个对象。

对第二个参数传递 `true` ，将使用 `querystring` 模块来解释查询字符串。如果为 `true` ，那么最后的对象中一定存在 `query` 属性，并且 `search` 属性将总是一个字符串（可能为空）。如果为 `false` ，那么 `query` 属性将不会被解释或解码。默认为 `false` 。

对第三个参数传递 `true` ，将会把 `//foo/bar` 解释为 `{ host: 'foo', pathname: '/bar' }` ，而不是 `{ pathname: '//foo/bar' }` 。默认为 `false` 。

url.format(urlObj)

接受一个解释完毕的URL对象，返回格式化URL字符串。

以下是格式化过程：

- `href` 将会被忽略。
- `path` 将会被忽略。
- 协议无论是否有末尾的冒号，都会被同样处理。
  - http，https，ftp，gopher，file协议的后缀是 `://` 。
  - 所有其他如mailto，xmpp，aim，sftp，foo等协议的后缀是 `:` 。

- 如果协议要求有 `://` ，**`slashes`** 会被设置为 **`true`**
    - 只有之前没有列出的要求有斜线的协议才需要被设置。如 `mongodb://localhost:8000/` 。
- `auth` 会被使用，如果存在的话。
- 只有当缺少 `host` 时，才会使用 `hostname` 。
- 只有当缺少 `host` 时，才会使用 `port` 。
- `host` 将会替代 `hostname` 和 `port` 。
- 无论有没有前导 / （斜线），`pathname` 都会被相同对待。
- 只有在缺少 `search` 时，才会使用 `query` （对象；参阅 `querystring` ）。
- **`search`** 将会替代 **`query`**
    - 无论有没有前导 ? （问号），它都会被相同对待。
- 无论有没有前导 # （英镑符号），`hash` 都会被相同对待。

url.resolve(from, to)

接受一个基础URL，和一个路径URL，并且带上锚点像浏览器一样解析他们。例子：

```
url.resolve('/one/two/three', 'four')         // '/one/two/four'
url.resolve('http://example.com/', '/one')    // 'http://example
.com/one'
url.resolve('http://example.com/one', '/two') // 'http://example
.com/two'
```

util

稳定度: 2 - 稳定

这些功能在模块 `'util'` 中，通过 `require('util')` 来使用它们。

`util` 模块主要的设计意图是满足 `node.js` 内部API的需要。但是许多工具对于你的程序也十分有用。如果你发现这些功能不能满足你的需要，那么鼓励你编写自己的工具集。我们对任何 `node.js` 内部功能不需要的功能，都不感兴趣。

util.callbackify(original)

> 于v8.2.0加入
>
> - original `async` 函数
> - Returns: 回调函数

参数接收一个 `async` 函数(或者函数返回值是 `promise`), 返回一个Node.js错误优先的回调函数, 第一个参数是错误原因(如果 `Promise` 处于 `resolve` , 则返回 `null`), 第二个参数返回结果

例子

```
const util = require('util');

async function fn() {
  return await Promise.resolve('hello world');
}
const callbackFunction = util.callbackify(fn);

callbackFunction((err, ret) => {
  if (err) throw err;
  console.log(ret);
});
```

输出:

```
hello world
```

备注:

- `callback` 是异步执行的，并且会有一个有限的堆栈跟踪。 如果 `callback` 抛出，进程将发出 `uncaughtException` 事件，如果不处理则退出。
- 由于 `null` 具有作为回调的第一个参数的特殊含义，因此如果外层函数拒绝带有虚假值的 `Promise` 作为原因(`reason`)，则该值将被包装在Error中，初识值将存储在名为 `reason` 的字段中。

util.debuglog(section)

> 于v0.11.3加入
>
> - section String 需要被调试的程序节点
> - Returns: Function 日志处理函数

这个方法被用来在 `NODE_DEBUG` 环境变量存在的情况下，创建一个有条件写入 `stderr` 的函数。如果 `section` 名出现在环境变量中，那么返回的函数与 `console.error()` 类似。否则，返回空函数。

例子：

```
var debuglog = util.debuglog('foo');

var bar = 123;
debuglog('hello from foo [%d]', bar);
```

如果程序在 `NODE_DEBUG=foo` 环境下运行，那么输出将是：

```
FOO 3245: hello from foo [123]
```

`3245` 是进程id。如果这个环境变量没有设置，那么将不会打印任何东西。

你可以通过逗号设置多个 `NODE_DEBUG` 环境变量。例如， `NODE_DEBUG=fs,net,tls` 。

util.deprecate(function, string)

> 于v0.8.0加入 `util.deprecate()` 方法包装给定的函数或类，使其被标记为已弃用。

```
const util = require('util');

exports.puts = util.deprecate(function() {
  for (let i = 0, len = arguments.length; i < len; ++i) {
    process.stdout.write(arguments[i] + '\n');
  }
}, 'util.puts: Use console.log instead');
```

当被调用时，util.deprecate()将返回一个函数，该函数将调用 `process.on('warning')` 事件发出一个 `DeprecationWarning` 警告。默认情况下，第一次被调用时，这个警告将被发射并打印到 `stderr` 。发出警告后，将调用包装函数。

如果 `--no-deprecation` 或 `--no-warnings` 其中一个命令行标记被使用，或者在第一个弃用警告之前将 `process.noDeprecation` 属性设置为 `true` ，那么 `util.deprecate()` 方法将不执行任何操作。

如果 `--trace-deprecation` 或 `--trace-warnings` 其中一个命令行标记被设置，或者将 `process.noDeprecation` 属性设置为 `true` ，当第一次调用 `deprecated` 方法时，会打印一个警告或者对战信息到 `stderr` 中。

如果 `--trace-deprecation` 命令被设置，或者将 `process.noDeprecation` 属性设置为 `true` ，在第一次调用 `deprecated` 方法时将会抛出异常

`-throw-deprecation` 命令行标记和 `process.throwDeprecation` 属性优先于 `--trace-depreation` 和 `process.traceDeprecation` 。

util.format(format[, ...])

使用第一个参数，像 `printf` 一样的格式输出格式化字符串。

第一个参数是一个包含了0个或更多占位符的字符串。每个占位符都被其后的参数所替换。支持的占位符有：

- %s - 字符串

- %d - 数字（整数和浮点数）
- %i - 整型
- $o - 对象。 是 `javascripts` 通用对象的一种字符串表现形式。
  与 `util.inspect()` 设置 `{ showHidden: true, depth: 4, showProxy: true }` 类似。将显示完成的对象，但不包括 `不可枚举的` 符号和属性。
- $O - 对象。 是 `javascripts` 通用对象的一种字符串表现形式。
  与 `util.inspect()` 不设置任何选项类似。将显示完成的对象，但不包括 `不可枚举的` 符号和属性。
- %j - JSON。如果参数包含循环引用，则返回字符串 `'[Circular]'` 。
- %% - 单独的百分比符号（ `'%'` ），它不消耗一个参数。

如果占位符没有对应的参数，那么占位符将不被替换。

```
util.format('%s:%s', 'foo'); // 'foo:%s'
```

如果参数多余占位符，那么额外的参数会被转换成字符串（对于对象和链接，使用 `util.inspect()` ），并且以空格连接。

```
util.format('%s:%s', 'foo', 'bar', 'baz'); // 'foo:bar baz'
```

如果第一个参数不是格式化字符串，那么 `util.format()` 将会返回一个以空格连接的所有参数的字符串。每一个参数都被调用 `util.inspect()` 来转换成字符串。

```
util.format(1, 2, 3); // '1 2 3'
```

util.inherits(constructor, superConstructor)

> - 历史
>   - v5.0.0 构造函数现在可以使用ES6中的 `类`
>   - v0.3.0 被添加

备注：不鼓励使用 `util.inherits()` 。请使用ES6类并扩展关键字以获得语言级别的继承支持。 另请注意，这两种风格在语义上是不兼容的。

- `constructor`
- `superConstructor`

将原型方法从一个构造函数继承到另一个构造函数。 构造函数的原型将被设置为由超级构造函数创建的新对象。

作为一种额外的便捷使用方法， `superConstructor` 将可以使用 `constructor.super_ property` 访问。

```
const util = require('util');
const EventEmitter = require('events');

function MyStream() {
  EventEmitter.call(this);
}

util.inherits(MyStream, EventEmitter);

MyStream.prototype.write = function(data) {
  this.emit('data', data);
};

const stream = new MyStream();

console.log(stream instanceof EventEmitter); // true
console.log(MyStream.super_ === EventEmitter); // true

stream.on('data', (data) => {
  console.log(`Received data: "${data}"`);
});
stream.write('It works!'); // Received data: "It works!"
```

在ES6中使用 `类` 和 `继承`

```
const EventEmitter = require('events');

class MyStream extends EventEmitter {
  write(data) {
    this.emit('data', data);
  }
}

const stream = new MyStream();

stream.on('data', (data) => {
  console.log(`Received data: "${data}"`);
});
stream.write('With ES6');
```

util.log(string)

在控制台输出带有时间戳的信息。

```
require('util').log('Timestamped message.');
```

util.inspect(object[, options])

返回一个代表了 `object` 的字符串，在调试时很有用。

一个可选的 `options` 对象可以被传递以下属性来影响字符串的格式：

- showHidden - 如果设置为 `true` ，那么对象的不可枚举属性也会被显示。默认为 `false` 。
- depth - 告诉 `inspect` 格式化对象时需要递归的次数。这对于巨大的复杂对象十分有用。默认为 `2` 。传递 `null` 表示无限递归。
- colors - 如果为 `true` ，那么输出会带有ANSI颜色代码风格。默认为 `false` 。颜色是可以自定义的，参阅下文。
- customInspect - 如果为 `false` ，那么定义在被检查对象上的 `inspect(depth, opts)` 函数将不会被调用。默认为 `false` 。

一个检查 `util` 对象所有属性的例子：

```
var util = require('util');

console.log(util.inspect(util, { showHidden: true, depth: null }));
```

参数值可以提供了它们自己的 `inspect(depth, opts)` 函数，当被调用时它们会收到当前的递归深度值，以及其他传递给 `util.inspect()` 的选项。

自定义 `util.inspect` 颜色

`util.inspect` 的有颜色的输出（如果启用）可以全局的通过 `util.inspect.styles` 和 `util.inspect.colors` 对象来自定义。

`util.inspect.styles` 是通过 `util.inspect.colors` 设置每个风格一个颜色的映射。高亮风格和它们的默认值为 `number (yellow) boolean (yellow) string (green) date (magenta) regexp (red) null (bold) undefined (grey) special` 。这时的唯一方法(cyan) * `name (intentionally no styling)` 。

预定义颜色有 `white, grey, black, blue, cyan, green, magenta, red` 和 `yellow` 。他们都是 `bold` ， `italic` ， `underline` 和 `inverse` 代码。

自定义对象的 `inspect()` 函数

对象也可以自己定义 `inspect(depth)` 函数， `util.inspect()` 将会调用它，并且输出它的结果：

```
var util = require('util');

var obj = { name: 'nate' };
obj.inspect = function(depth) {
  return '{' + this.name + '}';
};

util.inspect(obj);
  // "{nate}"
```

你也可以完全返回另一个对象，并且返回的字符串是由这个返回对象格式化而来的，这也 `JSON.stringify()` 相似：

```
var obj = { foo: 'this will not show up in the inspect() output'
 };
obj.inspect = function(depth) {
  return { bar: 'baz' };
};

util.inspect(obj);
  // "{ bar: 'baz' }"
```

util.inspect.custom

> 添加: v6.6.0 可用于声明定制检查功能的符号，可以参考Custom inspection functions on Objects

util.inspect.defaultOptions

> 添加: v6.4.0

`defaultOptions` 允许自定义 `util.inspect` 的默认选项。 这对于像 `console.log` 或 `util.format` 这样的函数是非常有用的，它隐式地调用 `util.inspect` 。 它应被设置为包含一个或多个有效的 `util.inspect()` 选项的对象。 当然直接设置选项属性也被支持的。

```
const util = require('util');
const arr = Array(101).fill(0);

console.log(arr); // 打印被截取过后的数组
util.inspect.defaultOptions.maxArrayLength = null;
console.log(arr); // 打印整个数组
```

util.promisify(original)

> v8.0.0加入

- `original`

- `Returns`

把Node.js的回调风格，即 `(err，value）=> ...` 作为最后一个参数，并返回一个 `promise` 。

举例:

```
const util = require('util');
const fs = require('fs');

const stat = util.promisify(fs.stat);
stat('.').then((stats) => {
  // Do something with `stats`
}).catch((error) => {
  // Handle the error.
});
```

或者使用 `async` :

```
const util = require('util');
const fs = require('fs');

const stat = util.promisify(fs.stat);

async function callStat() {
  const stats = await stat('.');
  console.log(`This directory is owned by ${stats.uid}`);
}
```

如果存在原生的 `[util.promisify.custom]` 属性， `promisify` 会返回执行后的结果，可以参考 `Custom promisified functions` 。

基本所有情况下， `promisify()` 方法都会假设 `original` 是将回调函数作为最后一个参数传入，如果不是，则返回 `undefined` 。

自定义 promisified 函数

使用 `util.promisify.custom` 符号可以覆盖 `util.promisify()` 的返回值：

```
const util = require('util');

function doSomething(foo, callback) {
  // ...
}

doSomething[util.promisify.custom] = function(foo) {
  return getPromiseSomehow();
};

const promisified = util.promisify(doSomething);
console.log(promisified === doSomething[util.promisify.custom]);
// prints 'true'
```

这对于源函数不遵循将错误优先回调作为最后一个参数的标准格式的情况很有用。

例如，这样一个函数 `(food，onSuccess Callback，onError Callback)`：

```
doSomething[util.promisify.custom] = function(foo) {
  return new Promise(function(resolve, reject) {
    doSomething(foo, resolve, reject);
  });
};
```

util.promisify.custom

> v8.0.0添加
>
> -

可用于声明函数的自定义 `promisified` 变体的符号，参考Custom promisified functions。

Class: util.TextDecoder

> v8.3.0添加 一种WHATWG编码标准TextDecoder API的实现。

```
const decoder = new TextDecoder('shift_jis');
let string = '';
let buffer;
while (buffer = getNextChunkSomehow()) {
  string += decoder.decode(buffer, { stream: true });
}
string += decoder.decode(); // end-of-stream
```

废弃掉的API

以下API已被弃用，不应再使用。 应该更新现有的应用程序和模块以寻找替代方法。

util._extend(target, source)

> v0.7.5添加 v6.0.0废弃 稳定度: 0 - 弃用: 用 `Object.assign()` 代替.

`util._extend()` 本来不打算在内部Node.js模块之外使用。 是社区的人找到他随意使用的。

它已被弃用，不应在新代码中使用。 `JavaScript` 通过 `Object.assign()` 提供了非常类似的内置功能。

util.isArray(object)

> 稳定度: 0 - 弃用

`Array.isArray` 的内部别名。

如果 `object` 是一个数组则返回 `true` ，否则返回 `false` 。

```
var util = require('util');

util.isArray([])
  // true
util.isArray(new Array)
  // true
util.isArray({})
  // false
```

util.isRegExp(object)

> 稳定度: 0 - 弃用

如果 `object` 是一个正则表达式则返回 `true` ，否则返回 `false` 。

```
var util = require('util');

util.isRegExp(/some regexp/)
  // true
util.isRegExp(new RegExp('another regexp'))
  // true
util.isRegExp({})
  // false
```

util.isDate(object)

> 稳定度: 0 - 弃用

如果 `object` 是一个日期则返回 `true` ，否则返回 `false` 。

```
var util = require('util');

util.isDate(new Date())
  // true
util.isDate(Date())
  // false (without 'new' returns a String)
util.isDate({})
  // false
```

util.isError(object)

> 稳定度: 0 - 弃用

如果 `object` 是一个错误对象则返回 `true` ，否则返回 `false` 。

```
var util = require('util');

util.isError(new Error())
  // true
util.isError(new TypeError())
  // true
util.isError({ name: 'Error', message: 'an error occurred' })
  // false
```

util.isBoolean(object)

> 稳定度: 0 - 弃用

如果 `object` 是一个布尔值则返回 `true` ，否则返回 `false` 。

```
var util = require('util');

util.isBoolean(1)
  // false
util.isBoolean(0)
  // false
util.isBoolean(false)
  // true
```

util.isNull(object)

> 稳定度: 0 - 弃用

如果 `object` 是严格的 `null` 则返回 `true` ，否则返回 `false` 。

```
var util = require('util');

util.isNull(0)
  // false
util.isNull(undefined)
  // false
util.isNull(null)
  // true
```

util.isNullOrUndefined(object)

> 稳定度: 0 - 弃用

如果 `object` 是一 `null` 或 `undefined` 则返回 `true` ，否则返回 `false` 。

```
var util = require('util');

util.isNullOrUndefined(0)
  // false
util.isNullOrUndefined(undefined)
  // true
util.isNullOrUndefined(null)
  // true
```

util.isNumber(object)

> 稳定度: 0 - 弃用

如果 `object` 是一个数字则返回 `true` ，否则返回 `false` 。

```
var util = require('util');

util.isNumber(false)
  // false
util.isNumber(Infinity)
  // true
util.isNumber(0)
  // true
util.isNumber(NaN)
  // true
```

util.isString(object)

> 稳定度: 0 - 弃用

如果 `object` 是一个字符串则返回 `true` ，否则返回 `false` 。

```
var util = require('util');

util.isString('')
  // true
util.isString('foo')
  // true
util.isString(String('foo'))
  // true
util.isString(5)
  // false
```

util.isSymbol(object)

> 稳定度: 0 - 弃用

如果 `object` 是一个 `Symbol` 则返回 `true` ，否则返回 `false` 。

```
var util = require('util');

util.isSymbol(5)
  // false
util.isSymbol('foo')
  // false
util.isSymbol(Symbol('foo'))
  // true
```

util.isUndefined(object)

> 稳定度: 0 - 弃用

如果 `object` 是 `undefined` 则返回 `true` ，否则返回 `false` 。

```
var util = require('util');

var foo;
util.isUndefined(5)
  // false
util.isUndefined(foo)
  // true
util.isUndefined(null)
  // false
```

util.isObject(object)

> 稳定度: 0 - 弃用

如果 `object` 严格的是一个对象而不是一个函数，则返回 `true` ，否则返回 `false` 。

```
var util = require('util');

util.isObject(5)
  // false
util.isObject(null)
  // false
util.isObject({})
  // true
util.isObject(function(){})
  // false
```

util.isFunction(object)

> 稳定度: 0 - 弃用

如果 `object` 是一个函数则返回 `true` ，否则返回 `false` 。

```
var util = require('util');

function Foo() {}
var Bar = function() {};

util.isFunction({})
  // false
util.isFunction(Foo)
  // true
util.isFunction(Bar)
  // true
```

util.isPrimitive(object)

> 稳定度: 0 - 弃用

如果 `object` 是一个基本值则返回 `true` ，否则返回 `false` 。

```
var util = require('util');

util.isPrimitive(5)
  // true
util.isPrimitive('foo')
  // true
util.isPrimitive(false)
  // true
util.isPrimitive(null)
  // true
util.isPrimitive(undefined)
  // true
util.isPrimitive({})
  // false
util.isPrimitive(function() {})
  // false
util.isPrimitive(/^$/)
  // false
util.isPrimitive(new Date())
  // false
```

util.isBuffer(object)

> 稳定度: 0 - 弃用

如果 `object` 是一个 `buffer` 则返回 `true` ，否则返回 `false` 。

```
var util = require('util');

util.isBuffer({ length: 0 })
  // false
util.isBuffer([])
  // false
util.isBuffer(new Buffer('hello world'))
  // true
```

util.inherits(constructor, superConstructor)

将一个构造函数所有的原型方法继承到到另一个中。构造函数的原型将会被设置为一个超类创建的新对象。

为了方便起见，超类可以通过 `constructor.super_` 来访问。

```
var util = require("util");
var events = require("events");

function MyStream() {
    events.EventEmitter.call(this);
}

util.inherits(MyStream, events.EventEmitter);

MyStream.prototype.write = function(data) {
    this.emit("data", data);
}

var stream = new MyStream();

console.log(stream instanceof events.EventEmitter); // true
console.log(MyStream.super_ === events.EventEmitter); // true

stream.on("data", function(data) {
    console.log('Received data: "' + data + '"');
})
stream.write("It works!"); // Received data: "It works!"
```

util.deprecate(function, string)

标记一个方法为不应再使用。

```
var util = require('util');

exports.puts = util.deprecate(function() {
  for (var i = 0, len = arguments.length; i < len; ++i) {
    process.stdout.write(arguments[i] + '\n');
  }
}, 'util.puts: Use console.log instead');
```

默认返回一个被运行时会发出一次警告的，修改后的函数。

如果 `--no-deprecation` 被设置，那么这个函数将为空。可以在运行时通过 `process.noDeprecation` 布尔值配置（只有在模块被加载前设置，才会有效）。

如果 `--trace-deprecation` 被设置，当被弃用的API第一次被使用时，会向控制台打印一个警告和堆栈信息。可以在运行时通过 `process.traceDeprecation` 布尔值配置。

如果 `--throw-deprecation` 被设置，那么当被弃用的API被使用时，应用会抛出一个错误。可以在运行时通过 `process.throwDeprecation` 布尔值配置。

`process.throwDeprecation` 的优先级高于 `process.traceDeprecation` 。

util.debug(string)

> 稳定度: 0 - 弃用: 使用 `console.error()` 代替。

被弃用， `console.error` 的前身。

util.error([...])

> 稳定度: 0 - 弃用: 使用 `console.error()` 代替。

被弃用， `console.error` 的前身。

util.puts([...])

> 稳定度: 0 - 弃用: 使用 `console.log()` 代替。

被弃用， `console.log` 的前身。

util.print([...])

> 稳定度: 0 - 弃用: 使用 `console.log()` 代替。

被弃用， `console.log` 的前身。

util.pump(readableStream, writableStream[, callback])

> 稳定度: 0 - 弃用: 使用 `readableStream.pipe(writableStream)` 代替。

被弃用， `stream.pipe()` 的前身。

V8

稳定度: 2 - 稳定

这个模块暴露了 `node.js` 内建的指定版本的V8的事件和接口。这些接口受上游（upstream）变化的影响，所以没有被稳定索引（stability index）所覆盖。

getHeapStatistics()

返回一个包含以下属性的对象。

```
{
  total_heap_size: 7326976,
  total_heap_size_executable: 4194304,
  total_physical_size: 7326976,
  used_heap_size: 3476208,
  heap_size_limit: 1535115264
}
```

setFlagsFromString(string)

设置额外的V8命令行标识。请谨慎使用；在虚拟机启动后改变设定可能会产生不可预测的行为，包括程序崩溃或数据丢失。或者它也可能什么都没有做。

当前 `node.js` 可用的V8选项，在运行 `iojs --v8-options` 命令的输出中显示。一个非官方，社区维护的配置列表：`https://github.com/thlorenz/v8-flags/blob/master/flags-0.11.md` 。

用处：

```
// Print GC events to stdout for one minute.
var v8 = require('v8');
v8.setFlagsFromString('--trace_gc');
setTimeout(function() { v8.setFlagsFromString('--notrace_gc'); }
, 60e3);
```

执行 `JavaScript`

稳定度: 2 - 稳定

要获取这个模块，你可以通过：

```
var vm = require('vm');
```

`JavaScript` 代码会被编译且立刻执行 或 编译，保存，并且稍后执行。

vm.runInThisContext(code[, options])

`vm.runInThisContext()` 编译代码，运行它，然后返回结果。运行中的代码不能访问本地作用域，但是可以访问当前的全局对象。

使用 `vm.runInThisContext` 和 `eval` 运行相同代码的例子：

```
var vm = require('vm');
var localVar = 'initial value';

var vmResult = vm.runInThisContext('localVar = "vm";');
console.log('vmResult: ', vmResult);
console.log('localVar: ', localVar);

var evalResult = eval('localVar = "eval";');
console.log('evalResult: ', evalResult);
console.log('localVar: ', localVar);

// vmResult: 'vm', localVar: 'initial value'
// evalResult: 'eval', localVar: 'eval'
```

`vm.runInThisContext` 不能访问本地作用域，所以 `localVar` 没有改变。 `eval` 可以访问本地作用域，所以 `localVar` 改变了。

这种情况下， `vm.runInThisContext` 更像是一个间接的 `eval` 调用，像 `(0,eval)('code')` 。但是，它还有以下这些额外的选项：

- filename: 允许你控制提供在堆栈追踪信息中的文件名。
- displayErrors: 是否在抛出异常前向 `stderr` 打印任何的错误，并且造成错误的行会被高亮。会捕获编译代码时的语法错误和编译完的代码运行时抛出的异常。默认为 `true` 。
- timeout: 在关闭之前，允许代码执行的时间（毫秒）。如果超时，一个错误被会抛出。

vm.createContext([sandbox])

如果指定了一个 `sandbox` 对象，则将 `sandbox` “上下文化”，这样它才可以被 `vm.runInContext` 或 `script.runInContext` 使用。在脚本内部， `sandbox` 将会是全局对象，保留了它自己所有的属性，并且包含内建对象和标准全局对象的所有函数。在由 `vm` 模块运行的脚本之外的地方， `sandbox` 将不会被改变。

如果没有指定 `sandbox` 对象，将会返回一个你可以使用的新的，无内容的 `sandbox` 对象。

这个函数在创建被用来运行多个脚本的沙箱时十分有用，例如，如果你正在模拟一个web浏览器，则可以创建一个代表了 `window` 全局对象的沙箱，然后在沙箱内运行所有的 `<script>` 标签。

vm.isContext(sandbox)

返回一个沙箱是否已经通过调用 `vm.createContext` 上下文化。

vm.runInContext(code, contextifiedSandbox[, options])

`vm.runInContext` 编译代码，然后将其在 `contextifiedSandbox` 中运行，然后返回结果。运行的代码不能访问本地作用域。 `contextifiedSandbox` 必须通过 `vm.createContext` 上下文化；它被用来当做代码的全局对象。

`vm.runInContext` 的选项和 `vm.runInThisContext` 相同。

例子：编译并执行不同的脚本，在同一个已存在的上下文中。

```
var util = require('util');
var vm = require('vm');

var sandbox = { globalVar: 1 };
vm.createContext(sandbox);

for (var i = 0; i < 10; ++i) {
    vm.runInContext('globalVar *= 2;', sandbox);
}
console.log(util.inspect(sandbox));

// { globalVar: 1024 }
```

注意，运行不受信任的代码是一个十分棘手的工作，需要十分小心。`vm.runInContext` 是十分有用的，但是为了安全的运行不受信任的代码，还是将它们放在另一个单独的进程中为好。

vm.runInNewContext(code[, sandbox][, options])

`vm.runInNewContext` 编译代码，接着，如果传递了 `sandbox` 则上下文化 `sandbox`，如果没有就创建一个新的已上下文化的沙箱，然后将沙箱作为全局对象运行代码并返回结果。

`vm.runInNewContext` 的选项和 `vm.runInThisContext` 相同。

例子：编译并执行一个 自增一个全局变量然后设置一个新的全局变量 的代码。这些全局变量包含在沙箱中。

```
var util = require('util');
var vm = require('vm');

var sandbox = {
  animal: 'cat',
  count: 2
};

vm.runInNewContext('count += 1; name = "kitty"', sandbox);
console.log(util.inspect(sandbox));

// { animal: 'cat', count: 3, name: 'kitty' }
```

注意，运行不受信任的代码是一个十分棘手的工作，需要十分小心。`vm.runInNewContext` 是十分有用的，但是为了安全的运行不受信任的代码，还是将它们放在另一个单独的进程中为好。

vm.runInDebugContext(code)

`vm.runInDebugContext` 编译代码，然后将它们在V8调试上下文中执行。主要的用途是访问V8调试对象：

```
var Debug = vm.runInDebugContext('Debug');
Debug.scripts().forEach(function(script) { console.log(script.name); });
```

注意，调试上下文和对象与V8的调试实现联系紧密，它们可能在没有事先提醒的情况就发生改变（或被移除）。

调试对象也可以通过 `--expose_debug_as= switch` 被暴露。

Class: Script

一个包含预编译代码，然后将它们运行在指定沙箱中的类。

new vm.Script(code, options)

创建一个编译代码但不执行它的新 `Script` 类。也就是说，一个创建好的 `vm.Script` 对象代表了它的编译完毕的代码。这个脚本已经通过下文的方法在晚些时候被调用多次。返回的脚本没有被绑定在任何的全局对象上。它可以在每次运行前被绑定，所以只在那次运行时有效。

`options` 可以有以下属性：

- filename: 允许你控制提供在堆栈追踪信息中的文件名。
- displayErrors: 是否在抛出异常前向 `stderr` 打印任何的错误，并且造成错误的行会被高亮。会捕获编译代码时的语法错误和运行时由脚本的方法的配置所控制的代码抛出的错误。

script.runInThisContext([options])

与 `vm.runInThisContext` 相似，但是是一个预编译的 `Script` 对象的方法。`script.runInThisContext` 运行脚本被编译完毕的代码，然后返回结果。运行中的代码不能访问本地作用域，但是可以访问当前的全局对象。

一个使用 `script.runInThisContext` 来编译一次代码，然后运行多次的例子：

```
var vm = require('vm');

global.globalVar = 0;

var script = new vm.Script('globalVar += 1', { filename: 'myfile.vm' });

for (var i = 0; i < 1000; ++i) {
  script.runInThisContext();
}

console.log(globalVar);

// 1000
```

`options` 可以有以下属性：

- displayErrors: 是否在抛出异常前向 `stderr` 打印任何的错误，并且造成错误的行会被高亮。只会应用于运行中代码的执行错误；创建一个有语法错误的 `Script` 实例是不可能的，因为构造函数会抛出异常。
- timeout: 在关闭之前，允许代码执行的时间（毫秒）。如果超时，一个错误被会抛出。

script.runInContext(contextifiedSandbox[, options])

与 `vm.runInContext` 相似，但是是一个预编译的 `Script` 对象的方法。 `script.runInContext` 运行脚本被编译完毕的代码，然后返回结果。运行中的代码不能访问本地作用域。

`script.runInContext` 的选项和 `script.runInThisContext` 相同。

例子：编译一段 自增一个全局对象并且创建一个全局对象 的代码，然后执行多次，这些全局对象包含在沙箱中。

```
var util = require('util');
var vm = require('vm');

var sandbox = {
  animal: 'cat',
  count: 2
};

var context = new vm.createContext(sandbox);
var script = new vm.Script('count += 1; name = "kitty"');

for (var i = 0; i < 10; ++i) {
  script.runInContext(context);
}

console.log(util.inspect(sandbox));

// { animal: 'cat', count: 12, name: 'kitty' }
```

注意，运行不受信任的代码是一个十分棘手的工作，需要十分小心。`script.runInContext` 是十分有用的，但是为了安全的运行不受信任的代码，还是将它们放在另一个单独的进程中为好。

script.runInNewContext([sandbox][, options])

与 `vm.runInNewContext` 相似，但是是一个预编译的 `Script` 对象的方法。如果传递了 `sandbox` , `script.runInNewContext` 将上下文化 `sandbox` ，如果没有就创建一个新的已上下文化的沙箱，然后将沙箱作为全局对象运行代码并返回结果。运行中的代码不能访问本地作用域。

`script.runInNewContext` 的选项和 `script.runInThisContext` 相同。

例子：编译一段 设置一个全局对象 的代码，然后在不同的上下文中多次执行它。这些全局对象包含在沙箱中。

```
var util = require('util');
var vm = require('vm');

var sandboxes = [{}, {}, {}];

var script = new vm.Script('globalVar = "set"');

sandboxes.forEach(function (sandbox) {
  script.runInNewContext(sandbox);
});

console.log(util.inspect(sandboxes));

// [{ globalVar: 'set' }, { globalVar: 'set' }, { globalVar: 'se
t' }]
```

注意，运行不受信任的代码是一个十分棘手的工作，需要十分小心。`script.runInNewContext` 是十分有用的，但是为了安全的运行不受信任的代码，还是将它们放在另一个单独的进程中为好。

Zlib

稳定度: 2 - 稳定

要获取这个模块，你可以通过：

```
var zlib = require('zlib');
```

它提供了 `Gzip/Gunzip` ， `Deflate/Inflate` 和 `DeflateRaw/InflateRaw` 类的绑定。每个类都有相同的选项，并且都是可读/可写流。

例子

可以通过将一个 `fs.ReadStream` 的数据导入一个 `zlib` 流，然后导入一个 `fs.WriteStream` ，来压缩或解压缩一个文件。

```
var gzip = zlib.createGzip();
var fs = require('fs');
var inp = fs.createReadStream('input.txt');
var out = fs.createWriteStream('input.txt.gz');

inp.pipe(gzip).pipe(out);
```

通过使用便捷方法，可以在一个步骤里完成压缩或解压缩数据。

```
var input = '.................................';
zlib.deflate(input, function(err, buffer) {
  if (!err) {
    console.log(buffer.toString('base64'));
  }
});

var buffer = new Buffer('eJzT0yMAAGTvBe8=', 'base64');
zlib.unzip(buffer, function(err, buffer) {
  if (!err) {
    console.log(buffer.toString());
  }
});
```

如果要在HTTP客户端或服务器上使用这个模块，在请求时需要带上 `accept-encoding` 头，在响应时需要带上 `content-encoding` 头。

注意，这些例子都只是非常简单的展示了一些基本的概念。 `zlib` 编码的开销是非常昂贵的，并且结果需要被缓存。更多关于速度/内存/压缩的权衡，请参阅下文的 `内存使用调优` 。

```
// client request example
var zlib = require('zlib');
var http = require('http');
var fs = require('fs');
var request = http.get({ host: 'izs.me',
                         path: '/',
                         port: 80,
                         headers: { 'accept-encoding': 'gzip,def
late' } });
request.on('response', function(response) {
  var output = fs.createWriteStream('izs.me_index.html');

  switch (response.headers['content-encoding']) {
    // or, just use zlib.createUnzip() to handle both cases
    case 'gzip':
      response.pipe(zlib.createGunzip()).pipe(output);
      break;
    case 'deflate':
```

```
      response.pipe(zlib.createInflate()).pipe(output);
      break;
    default:
      response.pipe(output);
      break;
  }
});

// server example
// Running a gzip operation on every request is quite expensive.
// It would be much more efficient to cache the compressed buffe
r.
var zlib = require('zlib');
var http = require('http');
var fs = require('fs');
http.createServer(function(request, response) {
  var raw = fs.createReadStream('index.html');
  var acceptEncoding = request.headers['accept-encoding'];
  if (!acceptEncoding) {
    acceptEncoding = '';
  }

  // Note: this is not a conformant accept-encoding parser.
  // See http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#
sec14.3
  if (acceptEncoding.match(/\bdeflate\b/)) {
    response.writeHead(200, { 'content-encoding': 'deflate' });
    raw.pipe(zlib.createDeflate()).pipe(response);
  } else if (acceptEncoding.match(/\bgzip\b/)) {
    response.writeHead(200, { 'content-encoding': 'gzip' });
    raw.pipe(zlib.createGzip()).pipe(response);
  } else {
    response.writeHead(200, {});
    raw.pipe(response);
  }
}).listen(1337);
```

zlib.createGzip([options])

根据一个 `options` ，返回一个新的 `Gzip` 对象。

zlib.createGunzip([options])

根据一个 `options` ，返回一个新的 `Gunzip` 对象。

zlib.createDeflate([options])

根据一个 `options` ，返回一个新的 `Deflate` 对象。

zlib.createInflate([options])

根据一个 `options` ，返回一个新的 `Inflate` 对象。

zlib.createDeflateRaw([options])

根据一个 `options` ，返回一个新的 `DeflateRaw` 对象。

zlib.createInflateRaw([options])

根据一个 `options` ，返回一个新的 `InflateRaw` 对象。

zlib.createUnzip([options])

根据一个 `options` ，返回一个新的 `Unzip` 对象。

Class: zlib.Zlib

这个类未被 `zlib` 模块暴露。它之所以会出现在这里，是因为它是 `compressor/decompressor` 类的基类。

zlib.flush([kind], callback)

`kind` 默认为 `zlib.Z_FULL_FLUSH` 。

冲刷等待中的数据。不要轻率地调用这个方法，过早的冲刷会给压缩算法带来消极影响。

zlib.params(level, strategy, callback)

动态地更新压缩等级和压缩策略。只适用于 `deflate` 算法。

zlib.reset()

将 `compressor/decompressor` 重置为默认值。只使用于 `inflate` 和 `deflate` 算法。

Class: zlib.Gzip

使用 `gzip` 压缩数据。

Class: zlib.Gunzip

解压一个 `gzip` 流。

Class: zlib.Deflate

使用 `deflate` 压缩数据。

Class: zlib.Inflate

解压一个 `deflate` 流。

Class: zlib.DeflateRaw

使用 `deflate` 压缩数据，不添加 `zlib` 头。

Class: zlib.InflateRaw

解压一个原始 `deflate` 流。

Class: zlib.Unzip

通过自动探测头信息，解压 `Gzip` 或 `Deflate` 压缩流。

便捷方法

所有的方法接受一个字符串或一个 `buffer` 作为第一个参数，并且第二个参数是一个可选的 `zlib` 类的配置，并且会以 `callback(error, result)` 的形式执行提供的回调函数。

每一个方法都有一个同步版本，除去回调函数，它们接受相同的参数。

zlib.deflate(buf[, options], callback)

zlib.deflateSync(buf[, options])

使用 `Deflate` 压缩一个字符串。

zlib.deflateRaw(buf[, options], callback)

zlib.deflateRawSync(buf[, options])

使用 `DeflateRaw` 压缩一个字符串。

zlib.gzip(buf[, options], callback)

zlib.gzipSync(buf[, options])

使用 `Gzip` 压缩一个字符串。

zlib.gunzip(buf[, options], callback)

zlib.gunzipSync(buf[, options])

使用 `Gunzip` 压缩一个字符串。

zlib.inflate(buf[, options], callback)

zlib.inflateSync(buf[, options])

使用 `Inflate` 压缩一个字符串。

zlib.inflateRaw(buf[, options], callback)

zlib.inflateRawSync(buf[, options])

使用 `InflateRaw` 压缩一个字符串。

zlib.unzip(buf[, options], callback)

zlib.unzipSync(buf[, options])

使用 `Unzip` 压缩一个字符串。

Options

每一个类都接受一个 `options` 对象。所有的 `options` 对象都是可选的。

注意一些选项只与压缩相关，会被解压缩类忽略：

- flush (默认： `zlib.Z_NO_FLUSH`)
- chunkSize (默认： `16*1024`)
- windowBits
- level (仅用于压缩)
- memLevel (仅用于压缩)
- strategy (仅用于压缩)
- dictionary (仅用于 `deflate/inflate` ，默认为空目录)

参阅 `http://zlib.net/manual.html#Advanced` 中 `deflateInit2` 和 `inflateInit2` 的描述来获取更多信息。

内存使用调优

来自 `zlib/zconf.h` ，将其修改为 `node.js` 的用法：

默认的内存要求（字节）为：

```
(1 << (windowBits+2)) +  (1 << (memLevel+9))
```

换言之： `windowBits=15` 的128K 加上 `menLevel = 8` （默认值）的128K 加上其他小对象的一些字节。

例子，如果你想要将默认内存需求从256K减少至128K，将选项设置为：

```
{ windowBits: 14, memLevel: 7 }
```

当然，它会降低压缩等级（没有免费的午餐）。

`inflate` 的内存需求（字节）为：

```
1 << windowBits
```

换言之：`windowBits=15`（默认值）的32K加上其他小对象的一些字节。

这是内部输出缓冲外的 `chunkSize` 大小，默认为16K。

`zlib` 压缩的速度动态得受设置的压缩等级的影响。高的等级会带来更好地压缩效果，但是花费的时间更长。低的等级会带来更少的压缩效果，但是更快。

通常，更高的内存使用选项意味着 `node.js` 会调用 `zlib` 更少次数，因为在一次单独的写操作中它可以处理更多的数据。所以，这是影响速度和内存占用的另一个因素。

常量

所有在 `zlib.h` 中定义的常量，都也被定义在了 `require('zlib')` 中。大多数操作中，你都将不会用到它们。它们出现在这里只是为了让你对它们的存在不套感到惊讶。该章节几乎完全来自 `zlib` 文件。更多详情请参阅 `http://zlib.net/manual.html#Constants` 。

允许的冲刷值：

```
zlib.Z_NO_FLUSH
zlib.Z_PARTIAL_FLUSH
zlib.Z_SYNC_FLUSH
zlib.Z_FULL_FLUSH
zlib.Z_FINISH
zlib.Z_BLOCK
zlib.Z_TREES
```

`compression/decompression` 函数的返回码。负值代表错误，正值代表特殊但是正常的事件：

```
zlib.Z_OK
zlib.Z_STREAM_END
zlib.Z_NEED_DICT
zlib.Z_ERRNO
zlib.Z_STREAM_ERROR
zlib.Z_DATA_ERROR
zlib.Z_MEM_ERROR
zlib.Z_BUF_ERROR
zlib.Z_VERSION_ERROR
```

压缩等级：

```
zlib.Z_NO_COMPRESSION
zlib.Z_BEST_SPEED
zlib.Z_BEST_COMPRESSION
zlib.Z_DEFAULT_COMPRESSION
```

压缩策略：

```
zlib.Z_FILTERED
zlib.Z_HUFFMAN_ONLY
zlib.Z_RLE
zlib.Z_FIXED
zlib.Z_DEFAULT_STRATEGY
```

`data_type` 域的可能值：

```
zlib.Z_BINARY
zlib.Z_TEXT
zlib.Z_ASCII
zlib.Z_UNKNOWN
```

`deflate` 压缩方法（当前版本只支持这一个）：

```
zlib.Z_DEFLATED
```

用于初始化`zalloc`，`zfree`，`opaque`：

```
zlib.Z_NULL
```